UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

RTR - Uma Abordagem Reflexiva para Programação de Aplicações Tempo Real

TESE SUBMETIDA À UNIVERSIDADE FEDERAL DE SANTA CATARINA PARA OBTENÇÃO DO GRAU DE DOUTOR EM ENGENHARIA, ÁREA ENGENHARIA ELÉTRICA, ÁREA DE CONCENTRAÇÃO SISTEMAS DE INFORMAÇÃO

Olinto José Varela Furtado

Florianópolis, 17 de novembro de 1997

RTR - UMA ABORDAGEM REFLEXIVA PARA PROGRAMAÇÃO DE APLICAÇÕES TEMPO REAL

OLINTO JOSÉ VARELA FURTADO

ESTA TESE FOI JULGADA ADEQUADA PARA OBTENÇÃO DO TÍTULO DE **DOUTOR EM ENGENHARIA**, ESPECIALIDADE ENGENHARIA ELÉTRICA, ÁREA DE CONCENTRAÇÃO SISTEMAS DE INFORMAÇÃO, E APROVADA EM SUA FORMA FINAL PELO PROGRAMA DE PÓS-GRADUAÇÃO

PROF. JEAN-MARIE FARINES, Dr.
(Orientador)

PROF. ADROALDO RAIZER, Dr.
(Coordenador do Curso)

BANCA EXAMINADORA:

PROF. JEAN-MARIE FARINES, Dr.
(presidente)

PROF. ROGÉRIO DRUMMOND, Ph.D.
(Relator)

PROF. JONI DA SILVA FRAGA, Dr.

PROFa. MARIA LÚCIA BLANCK LISBÔA, Dra.

PROF. PAULO ROBERTO FREIRE CUNHA, Ph. D.

PROF. VITÓRIO BRUNO MAZZOLA, Dr.

Para Mafalda, André e Henrique,
com muito amor

Agradecimentos

Aos Professores Jean-Marie Farines (orientador) e Joni da Silva Fraga (co-orientador), pela orientação, incentivo e dedicação durante o desenvolvimento deste trabalho.

Aos membros da banca, por terem aceitado julgar este trabalho e pelos valiosos comentários que fizeram.

Ao INE/CTC/UFSC, pelo afastamento concedido.

Aos amigos do INE, pelo incentivo e pela força recebidos.

Ao meu amigo Rômulo, pelo apoio e pelas frutíferas discussões durante todo o curso.

Aos professores, colegas e funcionários do LCMI e PGEEL, pelo apoio recebido e por esses bons anos de convívio.

Aos meus amigos e familiares, que de uma forma ou de outra contribuíram para obtenção deste título.

Aos meus pais, Hermes e Zenita, pela vida, pela educação, pelo amor e pelo encorajamento recebidos.

À minha esposa Mafalda e aos meus filhos André e Henrique, por todo o amor, o apoio e a compreensão ao longo de todos esses anos.

Abstract

Real-Time Systems differ from conventional computational systems because they need timeliness beyond correctness. Thus, models, languages and tools traditionally used in the development and programming of conventional systems are not appropriate for real-time systems since they do not have any support for time handling. Besides, the fast evolution of real-time applications determines the need of support for distribution, integration with non real-time systems and ability to adapt to changes in the system. Hence, the increasingly need for models and languages that allow the representation and the control of application's timing aspects in a flexible way, facilitate the management of system's complexity and reduce the dependence of specific operational supports.

This thesis proposes a model and a programming language that explore the potentialities of the object orientation and computational reflection paradigms, in order to contribute for the solution of several problems found nowadays in real-time systems programming. The proposed model, named RTR model, allows the definition and the use of timing constraints and scheduling algorithms according to the application needs and independent of the underlying runtime support, providing flexibility and portability. Besides, the separation among functional and control concerns resulting of the use of computational reflection, facilitates the management of complexity and increases the possibility of reusing and the maintenance capacity of the developed systems. The proposed language, named Java/RTR language, is an extension of the Java language that implements the RTR model, integrating its timing capacity with Java's conventional facilities.

The potentiality and expressiveness of the RTR approach are demonstrated through several examples involving different typical real-time situations, including the representation of the synchronization in multimedia applications. Besides, a distributed extension of the RTR model for open environment is described and exemplified. In addition, this thesis also presents a study about real-time models and languages based on objects and computational reflection.

Índice

Capítulo I - Introdução 1

I.1 - Motivação 1
I.2 - Objetivos da Tese 2
I.3 - Organização do texto 3

Capítulo II - Modelos e Linguagens de Programação Orientados a Objetos para Aplicações Tempo Real 5

II.1 - Introdução 5
II.2 - O uso de orientação a objetos e reflexão computacional no contexto tempo real ... 6
II.3 - Modelos de programação tempo real orientados a objetos 8
II.3.1 - Introdução 8
II.3.2 - Caracterização de modelos de programação tempo real 9
II.3.3 - Modelos orientados a objetos existentes 9
II.3.3.1 - Modelo "RTO.k" 9
II.3.3.2 - Modelo de objetos "RTC++" 11
II.3.3.3 - Modelo "DRO" 12
II.3.3.4 - Modelo de objetos "RealTimeTalk - RTT" 14
II.3.3.5 - Modelo "R^2" 15
II.3.3.6 - Real-Time Meta-Object Protocol (RT-MOP) 17
II.3.4 - Considerações finais sobre os modelos apresentados 18
II.4 - Linguagens de programação tempo real orientadas a objetos (LTROO) 19
II.4.1 - Introdução 19
II.4.2 - Requisitos e características das LTR 19
II.4.3 - LTROO existentes 21
II.4.3.1 - ADA95 22
II.4.3.2 - Real-Time C++ (RTC++) 23
II.4.3.3 - DROL 23
II.4.3.4 - RealTimeTalk (RTT) 25
II.4.3.5 - FLEX 26
II.4.3.6 - Real-Time Java (RT-Java) 28
II.4.4 - Comentários gerais sobre as linguagens apresentadas 30
II.5 - Análise do tempo de execução 32
II.5.1 - Introdução 32
II.5.2 - Métodos para obtenção do tempo de execução 32
II.5.2.1 - Medição do tempo de execução 32
II.5.2.2 - Cálculo do tempo de execução 33
II.5.3 - Visão geral das principais abordagens usadas para obtenção do TME 35
II.5.3.1 - Abordagem usada em RT-Euclid 36
II.5.3.2 - Abordagem usada no ambiente "MARS" 36
II.5.3.3 - Abordagem usada no ambiente "RTT" 37
II.5.3.4 - Abordagem usada em Real-Time Java 38
II.5.3.5 - Abordagem baseada em esquemas temporais 39

II.5.3.6 - Outras abordagens 39
II.5.4 - Considerações finais relativas ao cálculo do TME 41
II.6 - Conclusões 41

Capítulo III - O Modelo RTR 43

III.1 - Introdução 43
III.2 - Caracterização do Modelo RTR 43
III.3 - Estrutura geral e dinâmica de funcionamento 45
III.4 - Descrição detalhada 46
III.4.1 - Objetos-Base de Tempo Real (OBTR) 46
III.4.1.1 - Exemplo de um OBTR 49
III.4.2 - Meta-Objetos Gerenciadores (MOG) 50
III.4.2.1 - Seção de gerenciamento 50
III.4.2.2 - Seção de sincronização 53
III.4.2.3 - Seção de exceções temporais 53
III.4.2.4 - Seção de restrições temporais 53
III.4.2.5 - Exemplo de um Meta-Objeto Gerenciador 54
III.4.3 - Meta-Objeto Escalonador (MOE) 56
III.4.4 - Meta-Objeto Relógio (MOR) 58
III.5 - Explorando o potencial e demonstrando a expressividade do modelo RTR 59
III.5.1 - Introdução 59
III.5.2 - Refletindo aspectos não temporais 59
III.5.3 - Polimorfismo temporal 60
III.5.4 - Ajuste dinâmico do tempo de execução das tarefas 61
III.5.5 - Ajuste dinâmico dos atributos das restrições temporais 62
III.5.6 - Análise de escalonabilidade dinâmica 63
III.5.7 - Mudança dinâmica do algoritmo de escalonamento 64
III.5.8 - Uso simultâneo de diferentes políticas de escalonamento 64
III.5.9 - Controlando reflexivamente a disponibilidade de memória 65
III.6 - Expressando e implementando restrições de sincronização multimídia 66
III.6.1 - Relações de sincronização multimídia 67
III.6.2 - Exemplo de aplicação 69
III.7 - Uma extensão do modelo RTR para ambientes distribuídos abertos 74
III.7.1 - Aspectos básicos da extensão proposta 74
III.7.2 - Arquitetura CORBA 75
III.7.3 - Objetos-Base e Meta-Objetos de comunicação 76
III.7.4 - Cláusula timeout 77
III.7.5 - Comportamento da comunicação no modelo proposto 78
III.7.6 - Um exemplo de aplicação 79
III.7.7 - Protótipo da extensão proposta 83
III.8 - Comparação do modelo RTR com outros modelos 85
III.9 - Conclusões 87

Capítulo IV - Uma implementação do Modelo RTR : A Linguagem Java/RTR 88

IV.1 - Introdução 88
IV.2 - Considerações gerais sobre implementação do Modelo RTR 88

IV.2.1 - Abordagens para implementação da estrutura reflexiva do modelo 89
IV.2.2 - Estrutura e comportamento das aplicações 89
IV.2.3 - Concorrência e sincronização 90
IV.2.4 - Utilização de meta-objetos padrão 91
IV.2.5 - Uma implementação baseada na abordagem de simulação : Mapeamento do modelo RTR sobre a linguagem Java 91
IV.3 - A Linguagem Java/RTR 92
IV.3.1 - Introdução 92
IV.3.2 - Java - a linguagem-base escolhida 93
IV.3.2.1 - Visão geral de Java 93
IV.3.3 - Especificação de Java/RTR 95
IV.3.3.1 - Estrutura reflexiva de Java/RTR 95
IV.3.3.1.1 - Classes-base "Real-Time" (RTBC) 96
IV.3.3.1.2 - Meta-classes "Manager" (MMC) 97
IV.3.3.1.3 - Meta-Classes "Scheduler" (SMC) 98
IV.3.3.1.4 - Meta-Classes "Clock" (CMC) 99
IV.3.3.1.5 - Classes convencionais 100
IV.3.3.2 - Criação de objetos-base e meta-objetos 102
IV.3.3.3 - Facilidades temporais 102
IV.3.3.3.1 - Declaração de novos tipos de restrições temporais 103
IV.3.3.3.2 - Declaração de métodos com restrições temporais 103
IV.3.3.3.3 - Ativação de métodos com restrições temporais 106
IV.3.3.3.4 - Cláusula *timeout* 107
IV.3.3.4 - Interação entre objetos 107
IV.3.3.5 - Concorrência e sincronização 108
IV.3.3.6 - Distribuição 111
IV.3.4 - Implementação de Java/RTR 111
IV.3.4.1 - Introdução 111
IV.3.4.2 - O pré-processador Java/RTR 112
IV.4 - Análise do tempo de execução de programas Java/RTR 115
IV.4.1 - Considerações gerais sobre a abordagem proposta 117
IV.5 - Conclusões 118

Capítulo V - Conclusões 120

Referências bibliográficas 123

Apêndice A - Especificação sintática de Java/RTR 130

Apêndice B - Representação dos operadores de intervalo no modelo RTR 133

Índice de Figuras

Figura 2.1 - Estrutura geral do modelo R^2 15
Figura 2.2 - Relacionamento entre PERC e JAVA 30
Figura 3.1 - Estrutura geral do modelo RTR 45
Figura 3.2 - Exemplo de um Objeto-Base Tempo Real 49
Figura 3.3 - Dinâmica interna do Meta-Objeto Gerenciador relativa ao tratamento de (a) métodos com restrição temporal e (b) métodos sem restrição temporal. 51
Figura 3.4 - Exemplo de um Meta-Objeto Gerenciador 55
Figura 3.5 - Exemplo de implementação da restrição temporal "aperiodic" 56
Figura 3.6 - Estrutura geral de um Meta-Objeto Escalonador 57
Figura 3.7 - Estrutura geral de um Meta-Objeto Relógio 58
Figura 3.8 - Relações de intervalo básicas 67
Figura 3.9 - Operadores do modelo baseado em relações de intervalos estendido 67
Figura 3.10 - Representação gráfica da relação "Animation while(δ1, δ2) Audio" 68
Figura 3.11 - Tela de apresentação da aplicação Instrutor de Anatomia e Fisiologia 70
Figura 3.12 - Representação gráfica da sincronização entre as mídias que compõem a lição Músculo do Coração 71
Figura 3.13 - Representação da sincronização segundo o método baseado em intervalos 71
Figura 3.14 - Estrutura da solução proposta 71
Figura 3.15 - Pseudo-código do OBTR "Controlador" 72
Figura 3.16 - Pseudo-código do OBTR "Apresentador" 73
Figura 3.17 - Arquitetura CORBA 76
Figura 3.18 - Cláusula timeout 77
Figura 3.19 - Mecanismo para sincronização de estado em uma interação cliente/servidor 78
Figura 3.20 - Estrutura da extensão do Modelo RTR para ambientes distribuídos abertos 79
Figura 3.21 - Estrutura da aplicação 80
Figura 3.22 - Objeto-base e Meta-Objeto ClienteDeMidia 81
Figura 3.23 - Objeto-base e Meta-objeto ServidorDeMídia 82
Figura 3.24 - Interface IDL do objeto-base ServidorDeMídia 82
Figura 3.25 - Prioridades de escalonamento para os processos 83
Figura 3.26 - Prioridades de escalonamento para as threads 84
Figura 4.1 - Interface Protocol-MMC 98
Figura 4.2 - Interface Protocol-SMC 99
Figura 4.3 - Interface Protocol-CMC 100
Figura 4.4 - Transformação de uma classe convencional em uma classe RTBC 101
Figura 4.5 - Estrutura da cláusula timeout em Java/RTR 107
Figura 4.6 - Implementação de um buffer usando uma classe convencional Java 110
Figura 4.7 - Implementação de um buffer usando classes RTBC e MMC de Java/RTR 110

Capítulo I - Introdução

I.1 - Motivação

Sistemas Tempo Real (STR) são sistemas cuja correção deve ser temporal além de lógica; ou seja, são sistemas em que, diferentemente de sistemas computacionais convencionais, as restrições temporais impostas pelo ambiente devem ser consideradas e satisfeitas. Além de correção temporal ("timeliness") e previsibilidade ("predictability") que são requisitos básicos, os STR também apresentam requisitos funcionais comuns aos sistemas computacionais convencionais (não tempo real) complexos, tais como segurança de funcionamento ("dependability"), reusabilidade e capacidade de manutenção, os quais ganham conotação especial na presença do fator tempo [Stankovic 88] [Stoyenko 92] [Berryman 93]. Adicionalmente, os atuais e futuros STR requerem suporte para distribuição, integração com sistemas não tempo real e habilidade para adaptar-se a mudanças no sistema e no próprio ambiente operacional [Bosh 97] [Stankovic 96].

Em decorrência da evolução tecnológica, a demanda por Sistemas Tempo Real tem aumentado rapidamente nos últimos anos, tornando a tecnologia tempo real extremamente importante e necessária, na medida em que aplicações com requisitos temporais tornam-se mais comuns [Stankovick 96]. Dentre estas aplicações, destacam-se: sistemas militares, controle de tráfego aéreo, controle de processos industriais, sistemas automotivos, aplicações médicas, sistemas bancários, sistemas multimídia, e a maioria dos sistemas embarcados. Todas estas aplicações tem em comum a necessidade de correção temporal, embora diferenciem-se no tamanho, complexidade e severidade com que as restrições temporais devem ser consideradas e satisfeitas. Em função desta severidade, STR normalmente classificam-se em STR Hard, àqueles em que a satisfação das restrições temporais deve ser garantida (como por exemplo, sistemas militares de defesa) e STR Soft, àqueles em que a satisfação das restrições temporais pode ser submetida a uma política de melhor esforço (sistemas multimídia e sistemas bancários, por exemplo).

As especificidades dos STR decorrentes da necessidade de correção temporal, requerem que o fator tempo seja integralmente considerado em todos os estágios do processo de desenvolvimento (desde sua concepção até sua implementação) e seja suportado tanto pelo hardware e sistema operacional subjacentes quanto pela linguagem de programação utilizada para sua implementação. Adicionalmente são necessárias metodologias de desenvolvimento apropriadas e ferramentas de apoio específicas para análise do comportamento temporal dos programas tempo real produzidos.

Assim sendo, arquiteturas, sistemas operacionais, metodologias, linguagens de programação e ferramentas de apoio usadas tradicionalmente para o projeto, implementação e execução de sistemas computacionais convencionais, por não suportarem a noção de tempo, não se mostram adequadas ao desenvolvimento de STR, e conseqüentemente surge a necessidade de ferramentas específicas, apropriadas ao domínio tempo real.

Particularmente, com relação a modelos e linguagens para programação de aplicações tempo real - interesse básico desta tese, embora tenham sido propostas inúmeras extensões tempo real de linguagens convencionais (RTCC [Gehani 91], por exemplo) e mesmo novas linguagens especialmente projetadas para programação tempo real (RT-Euclid [Stoyenko 86,

91], por exemplo), o uso de linguagens convencionais, tais como assembly, C, e ADA tem predominado a prática de programação de STR nas últimas décadas. Contudo, excetuando-se alguns casos particulares, nenhuma destas abordagens tem se mostrado satisfatória frente as necessidades atuais da área.

Dentre os problemas básicos encontrados hoje na programação de STR, destacam-se os seguintes: dificuldade para gerenciar a complexidade inerente aos STR atuais, falta de flexibilidade na representação e controle dos aspectos temporais da aplicação, existência de um "gap" semântico entre projeto e implementação (decorrente da ausência de modelos de programação e metodologias apropriadas ao desenvolvimento de STR), além da forte dependência de ambientes operacionais (geralmente muito específicos), impedindo, na prática a existência de STR capazes de executar em ambientes de propósito geral (denominados STR abertos [Stankovick 96]), convivendo com outros STR desenvolvidos independentemente e mesmo com sistemas computacionais não tempo real.

Mais recentemente, em função das limitações das linguagens tempo real até então existentes e da inadequação das linguagens convencionais para programação tempo real, novos modelos (RTO.k [Kim 93, 94a,94b], DRO [Takashio 92, 93], R2 [Honda 94] e RT/MOP [Mitchell 97]) e linguagens (Flex [Lin 91], RTC++ [Ishikawa 90, 92], DROL [Takashio 92], RealTimeTalk [Gustafsson 94], Ada95 [Burns 96a, 96b] e Real-Time Java [Nilsen 95, 96a]) baseadas em tecnologias como distribuição, orientação a objetos e reflexão computacional, foram propostas e estão sendo desenvolvidas com o objetivo de melhor integrar aspectos funcionais e temporais e atender mais satisfatoriamente os requisitos de flexibilidade, reuso, manutenção, concorrência e distribuição. Entretanto, estas propostas ainda não resolvem completamente as necessidades da área, especialmente se considerarmos STR modernos e de grande escala (como por exemplo sistemas bancários, multimídia e CSCW), incluindo STR abertos, cujas características e complexidade são a motivação para a procura de novos modelos e linguagens.

I.2 - Objetivos da Tese

Inserido neste contexto, esta tese tem como objetivo propor um modelo para programação de aplicações tempo real capaz de contribuir na solução de vários dos problemas encontrados atualmente no desenvolvimento de STR (particularmente na programação de STR Soft), especialmente problemas relacionados com a representação dos aspectos temporais, com o gerenciamento da complexidade, com a necessidade de flexibilidade para especificação e controle dos aspectos temporais envolvidos, com o "gap" semântico existente entre projeto e implementação e com a dependência de ambiente operacional. Adicionalmente, também é proposta uma linguagem de programação tempo real que viabiliza o desenvolvimento de aplicações tempo real segundo a filosofia de programação introduzida pelo modelo.

O modelo proposto, denominado modelo RTR (Reflexivo Tempo Real), é um modelo de programação orientado a objetos, reflexivo e de tempo real, que caracteriza-se por permitir a representação e o controle de aspectos temporais (restrições, escalonamento e exceções) de aplicações tempo real que seguem uma abordagem de melhor esforço, de forma simples, flexível e natural.

A adoção de orientação a objetos permite a exploração das potencialidades do paradigma relativas a estruturação, manutenção, extensibilidade e reutilização, facilitando o gerenciamento da complexidade dos sistemas.

O uso de reflexão computacional (introduzido no modelo RTR através da abordagem de meta-objetos), permite que aspectos de controle relativos a restrições temporais, restrições de sincronização, exceções temporais e algoritmos de escalonamento tempo-real sejam implementados separadamente dos aspectos funcionais da aplicação, simplificando e propiciando uma maior flexibilidade no desenvolvimento de aplicações tempo-real, tornando-as mais legíveis, manuteníveis e reusáveis.

Além disso, o uso de reflexão permite que novas restrições temporais e algoritmos de escalonamento sejam definidos e utilizados sem nenhuma intervenção no código base da aplicação ou no suporte subjacente, favorecendo assim a evolução dos sistemas; da mesma forma, manipuladores de exceção temporal e mecanismo de sincronização podem ser modificados ou substituídos independentemente da aplicação. Assim sendo, a adoção de reflexão computacional vem de encontro à necessidade de novos mecanismos de estruturação mais efetivos para programação tempo real expressa no documento sobre direções estratégicas para pesquisa em computação tempo real [Stankovick 96], demonstrando a conformidade do modelo proposto com as tendências atuais na pesquisa de novas facilidades para programação tempo real.

A linguagem de programação proposta, denominada Java/RTR, é uma extensão da linguagem de programação Java (Sun Microsystems Inc.) que incorpora a estrutura reflexiva e a semântica de funcionamento do modelo RTR, permitindo a representação explícita e flexível dos aspectos temporais típicos de aplicações tempo real.

Conjuntamente, modelo e linguagem propostos constituem a base de um ambiente para programação de aplicações tempo real, particularmente adequado a classes de aplicações que suportam uma abordagem de melhor esforço, como por exemplo aplicações multimídia. Futuramente, este ambiente poderá vir a ser estendido com uma metodologia completa para o desenvolvimento de sistemas tempo real e com ferramentas para análise de comportamento temporal, reduzindo o "gap" semântico entre projeto e implementação de sistemas tempo real e permitindo a verificação sistemática da correção temporal dos sistemas desenvolvidos.

I.3 - Organização do texto

Esta tese está organizada em 5 capítulos. Neste primeiro capítulo apresentou-se a motivação básica para o desenvolvimento da tese, enfatizando-se a importância dos STR na atualidade e os principais problemas encontrados na programação desta classe de sistemas computacionais. Ainda neste capítulo, foram estabelecidos a composição e os objetivos da tese.

O capítulo II apresenta os principais aspectos relativos a modelos e linguagens de programação tempo real baseados em objetos, iniciando-se com uma discussão sobre o uso de orientação a objetos e reflexão computacional no desenvolvimento, e particularmente na programação, de sistemas tempo real. Em seguida, são descritos alguns dos principais modelos de programação tempo real baseados em objetos e/ou reflexão existentes, com ênfase nas contribuições e limitações destes modelos. Na sequência, após uma discussão preliminar sobre os requisitos e características das linguagens tempo real (LTR), são apresentadas e avaliadas algumas das principais LTR baseadas em objetos e/ou reflexão computacional existentes. Complementando o estudo sobre LTR são apresentados os principais aspectos relativos à análise do tempo de execução de programas tempo real, e descritas algumas abordagens existentes para cálculo do tempo de execução dos mesmos. Concluindo o capítulo, são discutidas as contribuições e limitações dos modelos e linguagens apresentados e explicitadas

as razões que nos levaram à proposição de um novo modelo e de uma nova linguagem para programação de aplicações tempo real.

No capítulo III, o modelo proposto, denominado modelo RTR, é apresentado. Inicialmente o modelo é caracterizado e sua concepção justificada; em seguida a estrutura geral do modelo é apresentada e seus componentes são detalhadamente descritos. Na sequência, a potencialidade e a expressividade do modelo são demonstradas através da identificação de várias situações tipicamente encontradas no desenvolvimento de sistemas tempo real que podem ser facilmente representadas pelo modelo RTR. Adicionalmente, é descrita uma proposta de extensão do modelo RTR para ambientes distribuídos abertos. Complementando o capítulo, é apresentada uma análise comparativa entre o modelo proposto e outros modelos existentes.

O capítulo IV descreve a linguagem de programação Java/RTR, proposta como parte complementar da presente tese. Inicialmente são discutidas algumas questões genéricas relativas a implementação do modelo. Em seguida, após os objetivos da linguagem desejada serem estabelecidos e as principais características de Java (a linguagem-base a ser estendida) serem apresentadas, a linguagem proposta (Java/RTR) é especificada. Nesta especificação a sintaxe e a semântica da extensão proposta são apresentadas e a integração destas extensões com a linguagem-base é comentada e exemplificada. Na seqüência, uma especificação preliminar do pré-processador destinado a traduzir programas Java/RTR para programas Java equivalentes é apresentada e a questão da análise do tempo de execução de programas Java/RTR é discutida. Complementando o capítulo, a linguagem proposta é comparada com outras LTR orientadas a objetos existentes.

No capítulo V são apresentadas as conclusões do trabalho, destacando-se as contribuições, as vantagens e as limitações da abordagem proposta. Finalizando o capítulo são descritas algumas perspectivas relativas à continuidade do trabalho.

Capítulo II - Modelos e Linguagens de Programação Orientados a Objetos para Aplicações Tempo Real

II.1 - Introdução

O desenvolvimento de Sistemas de Tempo Real (STR) envolve necessariamente a consideração de vários fatores tais como arquitetura, sistema operacional, metodologia de desenvolvimento, modelo e linguagem de programação e ferramentas para análise do comportamento temporal. Neste trabalho, entretanto, estamos particularmente interessados na programação de STR, e portanto nos ateremos nos aspectos diretamente relacionados a modelos e linguagens de programação.

Os requisitos das aplicações tempo real impõem um paradigma de programação bastante diferente da programação de sistemas convencionais. Enquanto na programação convencional a correção de um programa independe das características temporais de sua execução, um programa tempo real deve atender as restrições temporais da aplicação, requerendo correção nos domínios lógico e temporal.

Para construir programas que atendam estes propósitos, muitas linguagens de programação com características temporais baseadas em modelos de programação convencionais, foram e estão sendo desenvolvidas para programação tempo real [Stoyenko 92]. Contudo, a quase totalidade destas linguagens satisfaz apenas parcialmente ou de forma rudimentar a maioria dos requisitos que qualificam uma linguagem como sendo adequada para programação de STR (tais linguagens serão denominadas de Linguagem de programação Tempo Real - LTR). Apesar do esforço de pesquisa e dos muitos resultados positivos obtidos na última década, a área de programação tempo real ainda hoje continua defasada com relação às necessidades decorrentes da rápida evolução das aplicações tempo real, constituindo-se ainda em uma área aberta de pesquisa,.

Neste sentido, diversos modelos e linguagens de programação baseados em orientação a objetos, distribuição e reflexão computacional, tem sido propostos e desenvolvidos com o objetivo de tratar várias questões encontradas atualmente na programação de STR. Dentre estas, destacamos: o gerenciamento da complexidade; a flexibilidade na representação e no controle de aspectos temporais e no processo de desenvolvimento em geral; a reusabilidade; a manutenção e extensibilidade dos sistemas; a integração dos aspectos de concorrência, distribuição e tempo real; e a dependência de ambientes operacionais.

Este capítulo explora diferentes questões relacionadas à programação tempo real, e identifica as contribuições e limitações das propostas baseadas em objetos e/ou reflexão computacional existentes, com o objetivo de fundamentar e justificar a proposição de um novo modelo e de uma nova linguagem de programação tempo real, objetos deste trabalho. Inicialmente é discutido o uso de orientação a objetos e reflexão computacional no contexto da programação de aplicações tempo real. Em seguida, os principais modelos e linguagens existentes são apresentados e suas principais contribuições e limitações são enfatizadas. Complementando a discussão sobre linguagens tempo real, são apresentados os principais

aspectos relativos a obtenção e a análise do tempo de execução de programas tempo real e descritas algumas das principais abordagens usadas para este fim. Concluindo o capítulo, é constatada a necessidade de modelos e linguagens que explorem mais profundamente o potencial dos paradigmas de orientação a objetos e reflexão computacional na programação de sistemas tempo real.

II.2 - O uso de orientação a objetos e reflexão computacional no contexto tempo real

Orientação a objetos - O crescimento do tamanho e da complexidade de aplicações tempo real tem levado à busca de novas metodologias e ferramentas capazes de gerenciar mais eficientemente estes aspectos. O sucesso do paradigma de orientação à objetos no projeto e implementação de sistema computacionais convencionais [Meyer 88], tem influenciado a comunidade que atua na área de tempo real, a qual tem proposto modelos, metodologias de análise e projeto e linguagens de programação orientados à objetos.

O uso de objetos tem se mostrado bastante atrativo para programação tempo real, principalmente por facilitar o entendimento e o gerenciamento da complexidade dos STR; isto decorre do potencial do paradigma, cujas características básicas (estruturação em objetos, abstração, encapsulamento, herança e ligação dinâmica) favorecem os aspectos de modularidade, reuso, manutenção, flexibilidade e extensibilidade. Além disso, a adequação do paradigma de objetos no tratamento das questões de concorrência e distribuição é outro aspecto positivo.

Entretanto, por não suportar a representação e o controle dos aspectos temporais inerentes às aplicações tempo real, o paradigma de objetos - como concebido originalmente - não pode ser usado diretamente na modelagem/programação de STR, devendo antes ser estendido adequadamente para que os aspectos temporais possam ser efetivamente considerados.

Além disso, aspectos como ineficiência e falta de previsibilidade (normalmente atribuídos a sistemas orientados a objetos) levantam dúvidas sobre a adequação deste paradigma para sistemas tempo real. Contudo, estes aspectos devem ser vistos como deficiência das implementações existentes e não como um problema intrínseco do paradigma. No que diz respeito a falta de previsibilidade atribuída aos aspectos dinâmicos próprios do paradigma (ligação dinâmica e gerenciamento automático de memória, por exemplo), várias implementações recentes ([Gustafsson 94], [Bihari 92] e [Nilsen 96c]) apresentam soluções satisfatórias através do uso de técnicas deterministas. Entretanto, com relação a herança dinâmica e criação dinâmica de classes, a questão da previsibilidade permanece em aberto.

Enfim, embora restem dúvidas sobre a adequação do paradigma de objetos no desenvolvimento de STR *hard*, sua utilização na modelagem e programação de STR *soft*, apresenta-se como uma alternativa viável, cujo uso depende apenas da existência de modelos e linguagens tempo real que estendam o modelo de objetos, de forma que os aspectos temporais possam ser também considerados.

Reflexão computacional - Reflexão é a técnica pela qual um sistema pode "raciocinar" e atuar sobre si próprio. Os sistemas computacionais reflexivos contém dados que representam a estrutura e os aspectos computacionais do próprio sistema; desta forma, é

possível monitorar e modificar a estrutura e o comportamento do sistema através de computações realizadas pelo próprio sistema. Embora a computação reflexiva possa ser utilizada em qualquer paradigma de programação, seu uso é particularmente adequado no caso de orientação à objetos.

A abordagem comumente empregada para implementação de sistemas reflexivos orientados a objetos, proposta inicialmente em [Maes 87], tem sido a utilização de meta-objetos. Segundo esta abordagem, a cada objeto "x" é associado um meta-objeto "^x", o qual representa os aspectos estruturais e comportamentais de "x". Desta forma, a estrutura (reflexão estrutural) e o comportamento (reflexão comportamental) de "x" podem ser ajustados dinamicamente através de computações realizadas em "^x". Adicionalmente, como todo meta-objeto é também um objeto, ele pode ser manipulado como um objeto normal; desta forma, é possível a existência de um meta-meta-objeto de x (^^x), e assim sucessivamente.

A abordagem baseada em meta-objetos, permite separar os aspectos funcionais dos aspectos não funcionais, permitindo assim que a solução do problema em si (com relação as suas funcionalidades básicas) seja expressa através de objetos-base ("base-level objects") e que o controle do comportamento desses objetos (adaptando-os a um domínio específico) seja expresso através de meta-objetos ("meta-level objects").

A realização de reflexão computacional através da abordagem de meta-objetos introduz uma série de vantagens, dentre as quais podemos destacar:

- simplificação da programação dos aspectos funcionais, por permitir que o usuário abstraia-se dos aspectos de controle, concentrando-se exclusivamente nos seus aspectos algorítmicos;
- incremento da modularização, favorecendo a depuração e o entendimento dos sistemas.
- incremento no reuso e na manutenção de ambos, objetos-base e meta-objetos;
- incremento da flexibilidade do sistema, permitindo que diferentes estratégias de controle sejam aplicadas a diferentes objetos, de acordo com suas especificidades;
- incremento do controle do usuário sobre o sistema, permitindo que a estrutura e o comportamento dos objetos-base possam ser ajustados dinamicamente, via meta-objetos, de acordo com o estado do sistema;
- incremento da extensibilidade, favorecendo a evolução dos sistemas.

A reflexão computacional tem sido empregada nos últimos anos em áreas tais como : sistemas operacionais [Yokote 92], sistemas distribuídos [Chiba 93b] projeto de linguagens de programação [Kiczales 91], tolerância a faltas [Fraga 97b] [Fabre 95, 96] [Lisbôa 96] [Rubira 97], concorrência [Watanabe 88] [Masuhara 92], trabalho cooperativo [Dourish 96] e tempo real [Stankovic 93][Honda 94] [Mitchell 97].

Reflexão computacional e tempo real - Embora pouco explorada no domínio tempo real, a abordagem de reflexão computacional é vista como uma abordagem promissora no que se refere a estruturação de sistemas tempo real complexos [Stankovik 96], apta a contribuir nas questões de flexibilidade e gerenciamento da complexidade dos sistemas tempo real atuais e futuros. Além disso, sua adequação às questões correlatas a tempo real tais como distribuição, concorrência e tolerância a faltas, contribui para sua aplicação efetiva no desenvolvimento de STR. Além disso, o uso de reflexão no domínio tempo real, permite:

- adicionar ou modificar construções temporais (via meta-objetos) específicas de um domínio (ou de uma aplicação);
- definir um comportamento alternativo para o caso de exceções temporais (não satisfação de *deadlines*, por exemplo);
- substituir/alterar algoritmo de escalonamento, adequando-o a aplicação em questão;
- mudar o comportamento do programa, em função de informações obtidas em tempo de execução, como por exemplo disponibilidade de tempo, carga do sistema e atributos de *QoS (Quality of Service)*.
- definir e/ou ajustar o tempo de execução das atividades do sistema, em função da experiência adquirida com a evolução deste;
- possibilitar a realização de análise de escalonabilidade dinâmica, no meta-nível da aplicação;
- implementar protocolos relacionados à natureza tempo real;
- prover independência entre aplicação e ambiente operacional, através da implementação de controles (tipicamente realizados a nível de *runtime* ou sistema operacional) no meta-nível da aplicação;
- incrementar a portabilidade dos sistemas favorecendo sua utilização em ambientes abertos.

Entretanto, apesar do potencial da abordagem reflexiva, seu uso para tempo real é questionável com relação a dois importantes aspectos: desempenho e previsibilidade.

Com relação ao desempenho, admite-se um custo adicional devido ao processamento reflexivo. Entretanto, através de implementações adequadas este custo pode ser mantido a um nível aceitável; neste sentido, a adoção de implementações mistas (integrando objetos reflexivos e não-reflexivos) e o emprego de técnicas especiais (tais como avaliação parcial e *lazzy* avaliação) usadas em [Chiba 95], [Masuhara 92] e [Ruf 93], por exemplo, tem contribuído para a redução do *overhead* causado pela reflexão.

Com relação a previsibilidade, o problema não está na reflexão em si, mas sim no fato de que ela habilita a produção de sistemas tempo real muito mais flexíveis [Mitchell 97], para os quais a análise de pior caso ainda é uma questão de pesquisa em aberto; contudo, embora a questão de previsibilidade (especialmente na presença de reflexão estrutural) possa dificultar (ou mesmo impedir) o uso de reflexão na programação de STR *hard*, sua utilização na programação de STR *soft* (onde as exigências de previsibilidade são menos severas) é perfeitamente viável e promissora.

Finalmente, outro aspecto importante a salientar, é que o sucesso do uso de reflexão, independentemente do domínio da aplicação (mas particularmente no domínio tempo real), passa necessariamente pela correta identificação das informações a serem refletidas, pela proposição de arquiteturas (modelos) reflexivas bem definidas e por implementações adequadas (via linguagens de programação com suporte para reflexão) destas arquiteturas.

II.3 - Modelos de programação tempo real orientados a objetos

II.3.1 - Introdução

Os atuais e futuros STR, além de satisfazer requisitos específicos básicos como correção temporal, previsibilidade e segurança de funcionamento, devem também satisfazer

requisitos como modularidade, reusabilidade, manutenibilidade, distribuição, integração com sistemas não tempo real e flexibilidade. Em função disto, novos modelos de programação tempo real, baseados em orientação a objetos e reflexão computacional, tem sido propostos. Alguns destes modelos são definidos explicitamente, enquanto outros são definidos implicitamente em metodologias de desenvolvimento ou linguagens de programação.

Nesta seção, após caracterizarmos modelos de programação tempo real em geral e orientados a objetos em particular, descreveremos algumas das principais propostas de modelos baseados em objetos existentes, destacando suas principais características, contribuições e limitações.

II.3.2 - Caracterização de modelos de programação tempo real

Um modelo de programação para tempo real é uma abstração capaz de representar tanto o sistema computacional de controle a ser produzido, quanto as entidades do mundo real (o ambiente da aplicação) por ele controladas; assim sendo, ele deve apresentar uma estrutura e uma semântica de funcionamento capaz de representar integralmente tanto os aspectos funcionais quanto os aspectos temporais das aplicações tempo real. Além disso, modelos de programação tempo real devem ser vistos como o aspecto central no processo de desenvolvimento de STR, servindo como base para o estabelecimento de metodologias e linguagens de programação tempo real.

Um modelo de objetos para tempo real é um modelo de programação tempo real fundamentado no modelo de objetos convencional, acrescido de capacidades para manipular os aspectos temporais inerentes às aplicações tempo real. Fundamentalmente, um modelo de objetos tempo real deve suportar o encapsulamento, a abstração, a manipulação e o entendimento tanto dos aspectos funcionais quanto dos aspectos temporais dos objetos. Assim sendo, em adição às características próprias do modelo de objetos convencional, em um modelo de objetos tempo real deverão também ser consideradas as seguintes características:

- Restrições temporais devem poder ser expressas, controladas e garantidas;

- Concorrência, distribuição e um mecanismo para manipulação de exceções temporais devem ser suportadas pelo modelo;

- Um mecanismo de comunicação adequado a estruturação e a forma de interação dos objetos do modelo e que considere a existência de restrições temporais, deve ser suportado;

- A implementação do modelo (instâncias de classes, métodos e mensagens) e o mecanismo de suporte (*runtime*) associado, devem ter características temporais previsíveis.

A independência de ambientes operacionais e linguagens de programação específicas, é uma qualidade adicional que pode ser exigida destes modelos.

II.3.3 - Modelos orientados a objetos existentes

Nesta seção, apresentamos uma breve descrição dos principais modelos de programação tempo real baseados em objetos, enfatizando suas características básicas, vantagens e limitações.

II.3.3.1 - Modelo "RTO.k"

Características gerais - Proposto por Kim e Kopetz [Kim 93, 94a, 94b], o modelo RTO.k (Real-Time Object model) é uma extensão do modelo de objetos convencional que

tem como objetivo tanto a modelagem do sistema computacional de controle em si, quanto a modelagem do ambiente subjacente. O modelo RTO.k é independente da linguagem usada para especificar/programar STR, independente da forma como herança e outras características inerentes ao paradigma de objetos são suportadas e independente do mecanismo de comunicação (inter-objetos) utilizado.

O modelo RTO.k caracteriza-se por introduzir as seguintes extensões ao modelo de objetos convencional:

- distinção entre métodos ativados por tempo (*Spontaneous methods* - SpM's) e por mensagens (*Service Methods* - SvM's);

- existência de uma restrição básica de concorrência;

- associação de *deadlines* a ativação de métodos;

- dados de tempo real, os quais tornam-se inválidos após um intervalo de tempo especificado (denominado "duração máxima de validade").

Restrições temporais - Com relação às restrições temporais e a forma de explicitá-las, RTO.k é definido da seguinte forma:

1 - Restrições temporais dos métodos espontâneos - Na definição de métodos espontâneos (SpM's), as restrições temporais são especificadas através de AAC's (Autonomous Activation Condition), as quais especificam:

- O intervalo de tempo no qual o método pode ser ativado, definido por tempo de ativação (*start*) e tempo de desativação (*end*);

- A freqüência com que o método deverá ser ativado (para métodos periódicos);

- O tempo de *start* (mais cedo e mais tarde) no qual cada ativação deverá ocorrer;

- O *deadline* referente a cada ativação.

Cada SpM pode possuir várias AAC's com a restrição de que não haja sobreposição nos intervalos de ativação especificados. Adicionalmente, cada SpM especifica o modo de escalonamento a ser empregado, através das opções "*always*" (estático) ou "*if-demanded*" (dinâmico). No primeiro caso, o método será ativado de acordo com as condições especificadas nas AAC's em tempo de projeto (modo *default*) e no segundo caso a ativação do método dependerá de um pedido de outro método do mesmo objeto, explicitando as restrições temporais referentes a esta ativação, as quais deverão estar de acordo com o intervalo de ativação especificado em uma das AAC's do método em questão.

2 - Restrições temporais dos métodos de serviço - Na definição de métodos de serviço (SvM's) a única restrição temporal existente diz respeito ao tempo para conclusão da execução do método, a qual é especificada através do limite de tempo de execução (cláusula "*finish-within*") ou de um *deadline* (cláusula "*finish-by*").

Concorrência - A "restrição básica de concorrência" introduzida pelo modelo RTO.k, baseia-se no conceito de ODS (Object-Data Section) e tem como objetivo evitar conflitos entre SpM's e SvM's. Uma ODS é uma unidade de armazenamento atômica composta por definições de tipo e declaração de variáveis, e é usada para facilitar a detecção de conflitos entre unidades (cada método especifica as ODS's que irá utilizar, e a forma como o fará: *read-only* ou *read-write*) concorrentes.

Segundo esta restrição, SvM's só podem ser ativados quando não existir possibilidade de conflitos com SpM's, priorizando na prática a ativação de SpM's; contudo, o esquema

adotado não impõe qualquer restrição referente a execução concorrente entre SpM's e entre SvM's. Além disto, o modelo RTO.k estabelece que:

- A concorrência entre SpM's é especificada implicitamente quando da especificação destes métodos, como conseqüência da sobreposição de seus intervalos de ativação;

- A concorrência entre SvM's e entre SvM's e SpM's é permitida desde que não haja conflito entre as ODS's utilizadas pelos métodos envolvidos.

Comunicação - A interação entre objetos RTO.k se dá através de chamadas a partir de objetos clientes para métodos de serviço dos objetos servidores; o chamador pode ser tanto um SpM quanto um SvM. Visando facilitar operações concorrentes entre objetos clientes e servidores, as chamadas (requisições de serviço) para SvM's podem ser bloqueantes ou não-bloqueantes. No caso não-bloqueante o cliente não espera o resultado da operação e adicionalmente fornece uma lista de SvM's para os quais os resultados deverão ser enviados. A chegada de uma mensagem resultado em um SvM é tratada como um pedido de execução do referido SvM. Adicionalmente, SvM's podem chamar outros SvM's dentro de um mesmo objeto; SpM's só podem ser ativados a partir de métodos pertencentes ao mesmo objeto.

Implementação do modelo - O modelo RTO.k, dada a sua independência de linguagem e de ambiente de execução, nada especifica com relação a materialização de sua filosofia, ficando este trabalho para ser definido quando do refinamento do modelo visando uma implementação particular. Entretanto, em [Kim 94a, 94b] são especulados diferentes modelos de execução distribuídos, capazes de suportar aplicações de tempo real estruturadas segundo o modelo RTO.k. Além disto, uma implementação de RTO.k sobre o kernel DREAM, através de uma coleção de classes C++ foi proposta recentemente em [Kim 96].

Avaliação do modelo - Inicialmente, deve ser destacado no modelo RTO.k sua independência de ambientes operacionais e sua capacidade em modelar não só o sistema computacional de controle em si, como também o ambiente que se pretende controlar, de maneira uniforme e precisa. Além disso, a clara separação entre SvM's e SpM's, juntamente com a possibilidade de se definir em tempo de projeto as restrições temporais dos SpM's, contribui para a obtenção das garantias necessárias no contexto tempo real *hard*.

Por outro lado, o modelo RTO.k não flexibiliza a definição de restrições temporais a nível de aplicação, as quais são fixas e seu controle é dependente do ambiente operacional no qual o modelo vier a ser implementado. Além disso, questões como a proibição de ativação de SpM's de um objeto a partir de outros objetos e a necessidade de definição estática dos intervalos de ativação, embora justificáveis no contexto tempo real *hard*, dificultam a estruturação de sistemas e reduzem a capacidade de reuso. Adicionalmente, o modelo proposto não especifica um esquema de manipulação de exceções temporais e nada define com relação ao escalonamento de tarefas, aspecto este que fica na dependência da capacidade de escalonamento tempo real dos ambientes nos quais o modelo vier a ser implementado.

II.3.3.2 - Modelo de objetos "RTC++"

Estrutura dos objetos - O modelo RTC++, definido implicitamente na linguagem RTC++ [Ishikawa 90, 92], estende o modelo de objetos convencional para permitir a descrição de propriedades de tempo real e é um modelo "*multi-threaded*". Segundo este modelo, um STR é composto por objetos ativos com restrições temporais (denominados objetos de tempo real), que interagem através de passagem de mensagens (objetos ativos possuem uma ou mais *threads* de controle, as quais podem executar concorrentemente).

Neste modelo as threads são preemptivas, sendo que cada *thread* é responsável por um ou mais métodos; adicionalmente uma coleção de *threads* ("*thread-group*") pode ser responsável por um mesmo conjunto de métodos. O modelo suporta dois tipos de *threads*: "MASTER", usados para definir tarefas periódicas ("*time-triggered*") e "SLAVE", usadas para definir as demais atividades com restrições temporais; a especificação das *threads* é feita estaticamente (quando da definição do objeto), e tais definições podem ser herdadas por sub-classes da classe corrente.

Restrições temporais - As restrições temporais podem ser expressas tanto a nível de métodos como a nível de comandos e são especificadas explicitamente a nível de linguagem. Da mesma forma, exceções temporais podem ser associadas a todas as construções sujeitas a restrições temporais (métodos e comandos), e devem especificar as ações alternativas a serem realizadas, através da identificação de uma função (no caso de métodos) ou da especificação de um bloco de comandos (no caso de comandos).

Concorrência - O controle de concorrência é realizado através do uso de regiões críticas, as quais são implementadas através do comando *region*, o qual também pode estar sujeito a restrições temporais. Sincronização condicional é obtida pela associação de expressões guardas aos métodos de um objeto.

Comunicação - Com relação a comunicação, o modelo RTC++ suporta apenas comunicação síncrona (objetos comunicam-se da forma tradicional), permitindo adicionalmente uma segunda forma de "*reply*", no qual o "*sender*" pode continuar executando.

Implementação - Este modelo foi implementado através da linguagem RTC++, sobre o kernel ARTS [Tokuda 89] (na Carneggie Mellon University).

Avaliação do modelo - No modelo RTC++ destacam-se a naturalidade e a simplicidade relativas a especificação de restrições temporais e dos manipuladores de exceção. Diferentemente do modelo RTO.k, o modelo RTC++ não se propõe a ser uma base para modelagem completa de STR, mas sim estabelecer uma filosofia de programação para implementação de STR. O uso de objetos "*multiple thread*" é interessante tanto pelo fato de permitir uma representação mais fiel de entidades do mundo real, quanto pelo fato de incrementar o grau de concorrência dos sistemas.

Com relação as restrições temporais, o modelo é bastante rígido, não dando liberdade para a definição de novas restrições temporais, que apesar da simplicidade não possuem a mesma expressividade nem a mesma abrangência de outros modelos. Além disso, RTC++ não oferece nenhuma flexibilidade com relação ao escalonamento, o qual é realizado pelo suporte subjacente através da abordagem "rate monotonic".

Por outro lado, o modelo RTC++ é extremamente dependente de um ambiente operacional específico, dificultando sua implementação em ambientes de propósito geral. Além disso, o modelo carece de um mecanismo de comunicação assíncrona, para incrementar a concorrência do sistema e melhor adaptar-se a aplicações específicas. Adicionalmente, a existência de restrições temporais a nível de comando, fere a uniformidade do modelo de objetos e dificulta o reuso e a manutenção de programas.

II.3.3.3 - Modelo "DRO"

Características gerais - Proposto por Takashio e Tokoro [Takashio 92, 93], DRO é um modelo de objetos tempo real distribuído (mono-thread) que estende o modelo de objetos convencional para suportar computação tempo real através da encapsulação de restrições

temporais. O modelo DRO suporta as propriedades de melhor esforço ("*best-effort*") e menor prejuízo ("*least-suffering*") através do mecanismo de invocação polimórfica e da associação de restrições temporais aos métodos dos objetos.

Estrutura dos objetos - Uma característica fundamental do modelo DRO é a utilização de meta-objetos como forma de separar explicitamente questões funcionais (definidas nos objetos básicos) das questões não-funcionais (controle de concorrência, de tempo e de comunicação - definidas nos meta-objetos). Os meta-objetos introduzidos por DRO são: *AbstractStateMeta*, que gerenciam os conjuntos habilitados (usados para controlar concorrência) e a fila de mensagens; e *ProtocolMeta*, que define e controla os protocolos de comunicação (dando semântica para a comunicação inter-objetos) e gerencia as restrições temporais associadas a cada método invocado.

Restrições temporais - O modelo DRO suporta tanto a definição de tarefas periódicas ("*active methods*") quanto a especificação do tempo de execução de todos os métodos dos objetos de tempo real. Estas restrições temporais são associadas aos métodos dos objetos. Adicionalmente, o modelo pressupõe a existência de manipuladores de exceções tanto a nível de declaração como a nível de invocação de métodos, sendo que no caso de invocação também é previsto tratamento de exceção relativo a rejeição de uma invocação ("*reject*") e terminação anormal de um método invocado ("*abort*").

Comunicação - O mecanismo básico de interação entre objetos do modelo DRO é o polimorfismo temporal [Takashio 93], o qual pode ser obtido através das seguintes estratégias:

- Invocação de tempo polimórfico definida pelo servidor - consiste na declaração de uma função virtual associada a vários corpos com diferentes restrições temporais, um dos quais será selecionado pelo servidor (quando da ativação da função virtual), em função de sua disponibilidade de tempo no momento da invocação;

- Invocação de tempo polimórfico definida pelo cliente - (abordagem implementada pela linguagem DROL) - neste caso, o objeto servidor contém diversos métodos (com diferentes restrições temporais) para a mesma funcionalidade, e o cliente no momento da invocação deve fornecer as diversas alternativas aceitáveis, das quais uma será escolhida para execução, dependendo da restrição temporal expressa na invocação e da disponibilidade de tempo no servidor neste momento.

Adicionalmente, o modelo DRO suporta a definição de protocolos de comunicação por parte do usuário, a partir das primitivas assíncronas "*send*"/"*receive*". O protocolo a ser usado na invocação de um objeto deve ser explicitamente identificado, o qual determinará a semântica da invocação.

Concorrência - Em DRO, o controle de concorrência é realizado através do mecanismo de estados habilitados (cuja definição é similar a abordagem proposta em [Tomlinson 89]. Este mecanismo consiste na identificação dos possíveis estados em que um objeto pode encontrar-se, e na definição das transições que podem ser realizadas a partir de cada um desses estados com relação aos métodos dos objetos.

Implementação do modelo - O modelo DRO foi implementado sobre o kernel ARTS através da linguagem DROL;

Avaliação do modelo - Destaca-se no modelo DRO sua flexibilidade relativa a interação entre objetos, decorrente da possibilidade de definição de protocolos a nível de

programação e da introdução do mecanismo de invocação polimórfica temporal. Da mesma forma, a separação entre o controle da concorrência e das restrições temporais dos aspectos funcionais da aplicação (via filosofia de meta-objetos) contribui para o entendimento, manutenção e reusabilidade do *software* produzido.

Por outro lado, a capacidade reflexiva de DRO não é utilizada para definição e controle de novos tipos de restrições temporais, reduzindo a flexibilidade e a expressividade do modelo. Além disso, a questão de escalonamento tempo real não é considerada a nível de aplicação, sendo portanto dependente do ambiente operacional no qual o modelo está implementado.

II.3.3.4 - Modelo de objetos "RealTimeTalk - RTT"

Características gerais - RealTimeTalk [Eriksson 94] [Gustafsson 94] é um *framework* destinado ao desenvolvimento de aplicações tempo real, que caracteriza-se por assistir o projetista em todos os estágios do processo de desenvolvimento de STR, através de uma linguagem de descrição comum usada nos diferentes níveis de abstração deste processo. O principal objetivo de RTT é simplificar a modelagem e o projeto de sistemas tempo real previsíveis.

Estrutura dos objetos - O modelo de objetos RTT define como uma aplicação deverá ser configurada, e, ao contrário dos demais modelos aqui descritos, não encapsula as restrições temporais nos objetos, mas sim estabelece-as em um nível de abstração mais alto. Segundo este modelo, uma aplicação é decomposta em seus diferentes modos operacionais, juntamente com os modos-transição que especificam as atividades a serem realizadas quando ocorre uma transição de modo operacional.

Modos são refinados em um conjunto de "*usecase's*", os quais definem atividades específicas do sistema. Cada "*usecase*" é definido por um grafo de precedência e um grafo de relação de objetos, e possui um período associado; os grafos de relação de objetos são definidos com base nos PEO's (Parallel Executable Objects) que constituem um "*usecase*" e pelas relações existentes entre eles. Adicionalmente, objetos de interface e objetos de comunicação de dados também são visíveis em um grafo de relação de objetos. Os grafos de precedência definem a ordem na qual os métodos dos objetos serão escalonados; logo, o tempo de execução e o *deadline* de cada método deve ser definido. Tais grafos são especificados a partir dos PEO's envolvidos em uma atividade, juntamente com os atributos temporais destes PEO's. PEO's são os objetos básicos do modelo, os quais destinam-se a implementação das funcionalidades da aplicação.

Restrições e exceções temporais - No modelo RTT as restrições temporais não são especificadas explicitamente no corpo dos métodos dos objetos, e sim quando da definição dos grafos de precedência, onde cada tarefa é descrita por seu tempo de execução, seu *deadline* e seu "*release time*" relativo ao início do período especificado para o "*usecase*" que o grafo de precedência em questão está representando.

Exceções temporais em RTT podem ser definidas através de modos de exceção, que por sua vez podem ser definidos a nível de sistema (na fase de configuração do sistema) e a nível de modo, podendo ser reconfigurados em tempo de execução.

Outras características - No modelo RTT, a sincronização é especificada via grafos de precedência. Objetos RTT comunicam-se via passagem de mensagem, como no modelo convencional. Escalonamento em RTT é realizado *off-line*, por modo (o que significa que

cada modo e cada modo transição terá seu próprio "*schedule*") e com base nos grafos de precedência. As unidades de escalonamento (tarefas) são os métodos de ativação dos PEO's, os quais são não-preemptivos.

Avaliação do modelo - Um dos principais aspectos do modelo RTT é sua abrangência e uniformidade, envolvendo todas as fases do processo de desenvolvimento de STR. Aspectos estes que contribuem para a redução do "*gap*" semântico existente entre projeto e programação de sistemas.

Por outro lado, a especificação estática e em separado das restrições temporais e de precedência, embora adequada para STR *hard*, não satisfaz as necessidades de flexibilidade e integração que caracterizam a maioria dos STR *soft* e nem contribui para facilitar o gerenciamento da complexidade, o reuso e a manutenção destes sistemas. Além disso, o modelo proposto não é flexível com relação as questões temporais (restrições e escalonamento), cuja implementação é dependente do suporte de execução subjacente.

II.3.3.5 - Modelo "R^2"

Características gerais - Proposto por Honda e Tokoro [Honda 94], "R^2" (Real-time Reflective) é um modelo de computação concorrente baseado em objetos e reflexão computacional, segundo o qual um sistema computacional é estruturado como uma coleção de objetos básicos controlados por meta-objetos. Tais objetos comunicam-se através de passagem de mensagem (como no modelo convencional) e executam concorrentemente; entretanto, apenas um método (um fragmento de um *script*) pode executar por vez, visto que os objetos são "*single-threaded*".

O modelo proposto é voltado para sistemas de tempo real *soft*, visto que não existe garantia com relação a satisfação de restrições temporais; o que o modelo oferece é a garantia de que a violação de qualquer restrição temporal será detectado e que as ações alternativas especificadas serão realizadas.

Estrutura dos objetos - Além dos objetos básicos, o modelo R^2 introduz quatro tipos de meta-objetos: meta-objetos padrão, meta-objetos de tempo real, meta-objeto *alarm-clock* e meta-objeto escalonador. A interação entre os diversos tipos de objetos e meta-objetos que compõem o modelo R^2 é, esquematicamente, mostrada na figura 4.1 [Honda 94]. As funcionalidades destes objetos podem ser, resumidamente, assim descritas:

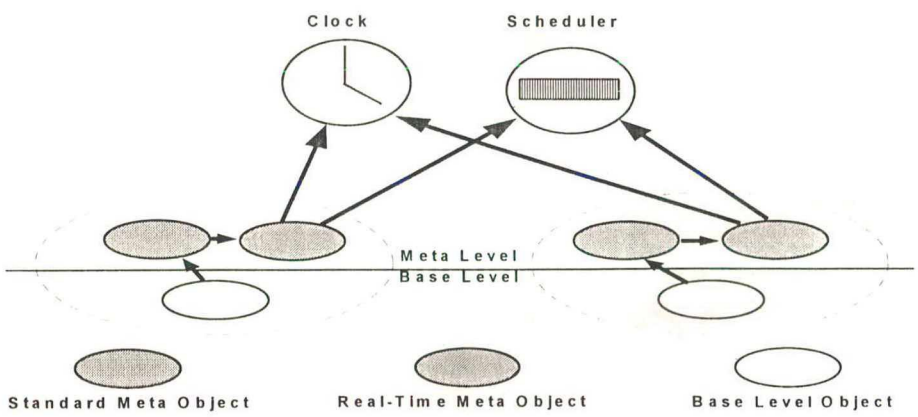


Figura 2.1 - Estrutura geral do modelo R^2

- Objetos Básicos - Seu *script* (correspondendo a uma coleção de métodos) implementa as funcionalidades da aplicação,

- Meta-Objeto Padrão - Responsável pelo controle de concorrência, pela manipulação de meta-mensagens relativas ao seu objeto básico e pelo suporte ao meta-objeto de tempo real correspondente.

- Meta-Objeto de Tempo Real - Processa as especificações temporais das tarefas interagindo com os meta-objetos *alarm-clock* e escalonador, visando prover as propriedades de melhor esforço e menor prejuízo. Também é responsável pela detecção de eventuais violações das restrições temporais especificadas e pela realização das ações alternativas apropriadas.

- Meta-Objeto *Alarm-Clock* - É uma abstração da função *clock* provida pelo sistema operacional/*hardware* subjacente, cujas funções básicas são fornecer o valor corrente do *clock* do sistema e notificar a passagem de um determinado intervalo de tempo.

- Meta-Objeto Escalonador - Determina a ordem de execução das tarefas (fragmentos de código do *script*), visando satisfazer as restrições temporais especificadas e melhorar a utilização da CPU. Na arquitetura proposta existe flexibilidade para escolha de um algoritmo de escalonamento que seja mais adequado à aplicação em questão.

Restrições temporais - as restrições temporais básicas consideradas no modelo são o tempo de *start* e o *deadline* de uma tarefa (uma seção de um *script*), as quais são associadas a segmentos de códigos dos objetos-base e reportadas para seu meta-objeto de tempo real que ficará responsável pelo processamento das restrições temporais especificadas. Além disso, o modelo proposto habilita a introdução de novas construções temporais, visando melhor se adequar às especificações temporais relativas a uma determinada aplicação.

Implementação do modelo - O modelo R^2 foi implementado como uma extensão da linguagem ABCL/R2, a qual é uma linguagem que incorpora os conceitos de concorrência de ABCL/1 [Yonezawa 86] e o conceito de reflexão individual de ABCL/R. Nesta implementação, os objetos e meta-objetos que compõem a arquitetura R^2 foram implementados como objetos de ABCL/R2.

Avaliação do modelo - O modelo R^2 introduz uma nova e interessante forma de desenvolvimento de STR usando a idéia de reflexão computacional baseada em meta-objetos. As principais vantagens decorrentes desta filosofia residem na flexibilidade e na expressividade que podem ser obtidas tanto pela definição de novas construções temporais quanto pela utilização de diferentes algoritmos de escalonamento (introduzidos via meta-objetos), sem afetar a programação das funcionalidades da aplicação e do suporte subjacente.

Apesar de ser claramente endereçado para STR *soft*, o modelo proposto empenha-se no sentido de realizar as propriedades de melhor esforço e menor prejuízo, buscando maximizar a taxa de satisfação das restrições temporais através do oferecimento de suporte para a definição de construções temporais e de algoritmos de escalonamento mais adequados a uma determinada classe de aplicações.

Um aspecto discutível do modelo R^2 é o fato de que as restrições temporais são associadas a segmentos de código e não a um "*script*"; desta forma, as restrições temporais especificadas não influenciarão o escalonamento dos "*script's*", mas somente o escalonamento de segmentos de código dos mesmos, reduzindo a efetividade do escalonamento realizado a nível de aplicação.

Adicionalmente, o esquema utilizado na expressão de restrições temporais dificulta a definição e o uso de restrições temporais do tipo *time-trigger* (como periodicidade, por exemplo), além de ferir a uniformidade do modelo de objetos e dificultar o reuso e a manutenção dos objetos-base.

Outro aspecto discutível no modelo R^2, é a não consideração do controle de sincronização condicional a nível de meta-objetos, uma vez que este aspecto pode afetar o escalonamento das tarefas do sistema. Além disso, o modelo R^2 é fortemente influenciado pelo paradigma de programação de ABCL/R (programação por continuação), de forma que a implementação de seu comportamento em outras linguagens, implicaria na mudança de sua filosofia e estruturação básicas.

II.3.3.6 - Real-Time Meta-Object Protocol (RT-MOP)

Características gerais - Em [Mitchell 97] é apresentada uma proposta inicial de um MOP (Meta-Object Protocol) de tempo real baseado em grupos de escalonamento, o qual tem como objetivo disciplinar e controlar mudanças em tempo de execução e servir como um mecanismo de estruturação de sistemas. Neste sentido o RT-MOP proposto pode ser visto como sendo um modelo de programação tempo real baseado em reflexão computacional.

Segundo o modelo proposto, um sistema tempo real deve ser estruturado na forma de objetos de aplicação (objetos-base) e meta-objetos, sendo que cada objeto da aplicação possui um meta-objeto associado, o qual é responsável pelo controle e pela modificação da estrutura e do comportamento do objeto de aplicação correspondente. Conjuntos de objetos de aplicação (e seus respectivos meta-objetos) são estruturados em grupos de escalonamento, sendo que cada grupo possui um meta-objeto "scheduler" responsável pelo escalonamento das tarefas de seus objetos constituintes segundo uma política de escalonamento própria; além disso, cada grupo possui também um gerente responsável pelo gerenciamento de entrada e saída de elementos no grupo (*membership*) em tempo de execução, sendo que o acesso a um grupo é controlado para manter garantias de escalonabilidade em tempo de execução. Adicionalmente, os meta-grupos de escalonamento também são reflexivos e podem ter seu comportamento alterado (mudança da política de escalonamento, por exemplo) através da invocação de operações do meta-meta-grupo.

No modelo proposto, nem todos os objetos da aplicação necessitam ser reflexivos, mas em qualquer caso as informações temporais necessárias ao escalonamento devem estar presentes na interface destes objetos; entretanto, na referência disponível [Mitchell 97], nada é adiantado sobre a forma pela qual tais informações são associadas aos objetos (ou a seus métodos...), e de que forma objetos não reflexivos serão considerados no escalonamento realizado pelo escalonador do grupo.

Concorrência e sincronização - O modelo de concorrência proposto, adota o conceito de objetos ativos formados pelo encapsulamento de tarefas (*threads*) dentro dos objetos; tal modelo é fortemente influenciado pelo modelo de concorrência usado em TAO [Mitchell 96]. Cada objeto ativo pode possuir várias *threads*, as quais são criadas quando o objeto é criado e são executadas de acordo com o escalonador do grupo de escalonamento ao qual o objeto pertence. Este modelo suporta tanto concorrência externa quanto concorrência interna, as quais são controladas através do uso de métodos sincronizados, conforme proposto em [Mitchell 96].

Distribuição - A questão de como os grupos serão distribuídos em uma rede ainda é uma questão em aberto; entretanto os autores consideram o grupo como sendo a unidade adequada para distribuição, e propõem que grupos de escalonamento dentro de um sistema possam residir em diferentes nodos da rede, não permitindo que os grupos possuam distribuição interna (visando tornar o escalonamento tempo real mais fácil).

Avaliação do modelo - Embora a proposta seja preliminar e muitos aspectos ainda estão por ser definidos (na referência considerada [Mitchell 97], apenas hipóteses são consideradas), o uso de grupos de escalonamento é sem dúvidas uma boa alternativa para prover a flexibilidade, de forma controlada, que muitos sistemas tempo real atuais necessitam; além disso, por ser reflexivo, o modelo proposto herda as vantagens do uso do paradigma de reflexão computacional para aplicações tempo real, conforme descrito na seção II.2 deste trabalho. Contudo, a realização prática deste modelo depende da definição de uma estrutura reflexiva concreta no âmbito de alguma linguagem de programação.

II.3.4 - Considerações finais sobre os modelos apresentados

Na presente seção foi enfatizado a necessidade de modelos apropriados para programação de sistemas tempo real; em seguida, modelos de programação tempo real em geral e modelos de objetos tempo real em particular foram caracterizados. Na seqüência, foram apresentados e comentados alguns dos principais modelos de objetos tempo real disponíveis na literatura. Encerrando esta seção, sintetizaremos algumas características comuns a maioria dos modelos estudados. Estas características, juntamente com algumas particularidades dos modelos considerados, servirão como ponto de partida para a definição do modelo de objetos tempo real a ser apresentado no próximo capítulo desta tese.

Características comuns aos vários modelos - Embora cada modelo apresentado tenha filosofia própria e conseqüentemente várias características específicas, alguns aspectos básicos são comuns a todos (ou pelo menos à maioria) os modelos estudados.

O primeiro ponto comum a ser destacado é a fidelidade estrutural e filosófica relativa ao modelo de objetos convencional; ou seja, todos os modelos estudados mantém a estrutura original dos objetos (baseados em métodos e mensagens) e caracterizam-se como modelo OTR através de extensões ao modelo convencional.

A independência relativa a características específicas do modelo convencional tais como herança, ligação dinâmica e *garbage collection*, é outro aspecto comum aos vários modelos; da mesma forma é comum a suposição de que tais características, se necessárias, devem ser consideradas a nível de definição das linguagens que vierem a implementá-los.

A maioria dos modelos apresentados (RTO.K e RT-MOP são exceções) estão intimamente relacionados a uma linguagem de programação, sendo que em alguns casos torna-se difícil a separação de ambos, haja visto que modelo e linguagem são definidos conjuntamente.

Também é comum entre os vários modelos apresentados, a representação explícita de restrições temporais; entretanto, com exceção de R^2, todos os demais modelos suportam apenas um conjunto fixo e pré-definido de restrições temporais, as quais são controladas pelo suporte subjacente. Com relação ao escopo, a maioria dos modelos suportam restrições temporais a nível de métodos, sendo que alguns modelos (RTC++ e DRO, por exemplo) suportam adicionalmente restrições temporais a nível de comandos enquanto que o modelo R^2 suporta exclusivamente restrições a nível de comandos.

Com relação ao escalonamento, com exceção de R^2, todos os demais modelos apresentados suportam apenas uma abordagem de escalonamento, a qual é fixa e dependente do ambiente operacional subjacente.

A falta de flexibilidade relativa a representação e controle de restrições temporais e a escolha de abordagens de escalonamento, característica da maioria dos modelos apresentados (R^2 é uma exceção), parece pouco adequada se considerarmos a dinâmica da área e a necessidade de integração, de evolução e de independência de ambiente operacional de muitos dos STR atuais. Neste sentido, a abordagem de reflexão computacional apresenta-se como uma alternativa promissora para a flexibilização destes aspectos.

Concorrência é também uma preocupação comum aos vários modelos; entretanto, a forma como ela é suportada e controlada difere substancialmente de um modelo para outro. Com relação a comunicação apenas o modelo DRO propõe um mecanismo especialmente projetado para o domínio tempo real; nos demais modelos, passagem de mensagem convencional é o mecanismo básico, embora alguns modelos permitam adicionalmente o uso de outras formas de comunicação.

Com exceção do modelo RTO.k, os modelos apresentados estabelecem algum mecanismo relativo ao tratamento de exceções temporais, embora em nenhum deles os limites de um objeto possam ser extrapolados (isto é, o tratamento é local ao objeto onde foi levantada a exceção).

Distribuição, embora não seja suportada por todos os modelos, é considerada uma característica indispensável para o domínio tempo real e está incluída nos planos de extensão dos modelos que ainda não a suportam. Por outro lado, nos modelos DRO e RTO.k, a distribuição foi considerada como um objetivo de projeto.

II.4 - Linguagens de programação tempo real orientadas a objetos (LTROO)

II.4.1 - Introdução

Linguagens tempo real orientadas a objetos (LTROO) apresentam-se como uma alternativa às linguagens tempo real (LTR) convencionais, para a solução de vários problemas encontrados atualmente na programação de aplicações tempo real.

Nesta seção, identificaremos os principais requisitos e características das LTR em geral, e descreveremos algumas das principais LTROO existentes, destacando suas características, vantagens e limitações. Para um estudo completo sobre LTR, recomenda-se as seguintes referências: [Stoyenko 92], [Halang 90], [Berryman 93], [Burns 96b] e [Furtado 95].

II.4.2 - Requisitos e características das LTR

Para que uma linguagem de programação seja apropriada para implementação de STR, ela deve satisfazer tanto os requisitos específicos ao contexto tempo real quanto os requisitos comuns a linguagens de propósito geral [Stoyenko 92], [Halang 90] e [Berryman 93], os quais ganham nova conotação em função da consideração dos aspectos temporais. Em outras palavras, uma LTR deve refletir os requisitos dos STR, incorporando a nível de linguagem e/ou suporte de execução características que permitam a satisfação destes requisitos. Nesta

seção, apresentamos de forma sucinta estes requisitos e destacamos as características necessárias para a satisfação dos mesmos.

Análise de escalonabilidade - Este requisito refere-se a capacidade de se prever antecipadamente se as tarefas de um sistema poderão ou não ser escalonadas de forma que suas restrições temporais sejam satisfeitas. Embora seja fundamental em STR *hard*, a importância da análise de escalonabilidade em STR *soft* é relativamente menor, na medida em que outros requisitos tais como flexibilidade, distribuição e capacidade de integração com componentes não tempo real ganham importância. Contudo, mesmo nestes casos, previsibilidade e correção temporal continuam sendo importantes e, na medida do possível, as LTR usadas na programação destes sistemas devem permitir, ainda que dinamicamente, a análise de escalonabilidade das partes tempo real do sistema.

As características desejáveis para que uma LTR permita a realização efetiva de análise de escalonabilidade, são as seguintes:

1 - Mecanismos ou construções que possibilitem a associação explícita de restrições temporais às construções da linguagem. Segundo [Halang 90], somente quando a dimensão "tempo" é incorporada explicitamente na linguagem, as restrições temporais podem ser expressas adequada e convenientemente.

2 - Construções com tempo de execução previsível e limitado; ou seja, é desejável que o uso de estruturas dinâmicas, recursões, *loops* ilimitados e transações sem *timeout* (bem como quaisquer outras construções que impeçam o comportamento previsível de um programa e conseqüentemente inviabilizem o cálculo do tempo de execução das tarefas), seja proibido ou pelo menos limitado.

3 - Os manipuladores de exceções temporais associados às construções temporais, também devem ter comportamento temporal previsível.

4 - É também desejável que as construções temporais da linguagem tenham atributos suficientes para permitir a realização do escalonamento segundo a(s) política(s) especificada(s);

Enfim, é importante ressaltar que a presença destas características a nível de linguagem por si só não são suficientes, para que a análise de escalonabilidade possa ser efetivamente realizada, é indispensável que o *hardware* e o *software* de suporte subjacentes também apresentem comportamento temporal previsível.

Robustez - Comum a todo sistema computacional, este requisito torna-se fundamental para STR, principalmente quando as falhas podem ser catastróficas. Para satisfazer este requisito, é desejável que a linguagem apresente as seguintes características : construções estruturadas, modularidade, tipagem forte e suporte para tratamento de exceções (funcionais e temporais). Adicionalmente, a existência de uma semântica clara e não-ambígua destas e das demais construções da linguagem, também contribui para este propósito.

Suporte a "programming-in-the-large" - Também comum a todas as linguagens endereçadas para programação de sistemas convencionais, este requisito é particularmente importante no domínio tempo real, em função do tamanho e da complexidade dos atuais e futuros STR. Visando melhor gerenciar o tamanho e a complexidade destes sistemas, é desejável que uma LTR, em adição a modularidade, suporte compilação em separado.

Suporte a concorrência - A linguagem deve permitir que a concorrência inerente às aplicações tempo real possa ser representada e controlada a nível de linguagem de programação. Assim sendo, a linguagem deve possuir mecanismos que permitam a definição,

sincronização e comunicação de atividades concorrentes, sem comprometer os aspectos temporais a elas associados.

Acesso direto ao *hardware* - *Software* de tempo real normalmente interage com processos físicos do mundo real, o que implica na necessidade de interfaceamento com uma grande variedade de dispositivos *hardware* (convencionais e de propósito especial). Por isto, é desejável que a linguagem utilizada possua facilidades para acesso ao *hardware* e manipulação de interrupções.

Manutenibilidade - Grande parte do custo associado a STR esta relacionado à sua manutenção. Portanto, é fundamental que programas tempo real sejam fáceis para entender, ler e modificar. Para isto, a linguagem deve ser simples, pequena e bem definida (se possível formalmente), provendo um alto grau de legibilidade. Além disso, uma boa estrutura modular juntamente com um sistema de tipo coerente e consistente contribui significativamente para o sucesso das atividades de manutenção.

Reusabilidade - Na medida em que os STR tornam-se cada vez maiores e mais complexos, a possibilidade de reutilização de componentes já testados não só incrementa a produtividade como também aumenta a confiabilidade no sistema produzido. Modularidade, facilidades relativas ao uso de bibliotecas, separação entre aspectos funcionais e de controle e herança, são mecanismos que favorecem a reusabilidade, e que portanto devem estar presentes nas linguagens destinadas a programação de STR.

Ambiente e suporte de execução - Além dos mecanismos e construções acima mencionados que contribuem para a caracterização de linguagens tempo real, Halang, em [Halang 90], enfatiza a necessidade de mecanismos e serviços a nível de suporte ("*run-time*"), que garantam a previsibilidade e a confiabilidade expressas a nível de linguagem, bem como ferramentas que viabilizem a análise e verificação do *software* implementado.

Flexibilidade - Embora flexibilidade oponha-se a previsibilidade, cada vez mais os STR atuais exigem soluções de compromisso entre estes requisitos. A necessidade de flexibilidade decorre de vários fatores, dentre os quais podemos destacar a necessidade de adição/remoção dinâmica de partes tempo real de uma aplicação e também a possibilidade de mudanças dos requisitos temporais da aplicação em função do estado do sistema [Bosch 97]. Além disto, flexibilidade também é importante para permitir a adequação de uma linguagem a diferentes classes de aplicação (por exemplo, através da definição e uso de diferentes tipos de restrições temporais e algoritmos de escalonamento) e/ou diferentes ambientes operacionais, e também em função da necessidade crescente de integração entre STR desenvolvidos independentemente e que executam em ambientes de propósito geral.

Para prover esta flexibilidade, além de uma boa estrutura modular, uma linguagem deve possuir mecanismos adequados para criação/modificação dinâmica de componentes e para controle dinâmico do comportamento destes componentes; particularmente, muitas das características inerentes aos paradigmas de orientação a objetos (polimorfismo por exemplo) e reflexão computacional (separação de propósitos, por exemplo) parecem extremamente adequadas para a satisfação deste requisito.

II.4.3 - LTROO existentes

Esta seção tem como objetivo descrever as principais LTR orientadas a objetos (LTROO) existentes, visando registrar o estudo realizado sobre LTR e ao mesmo tempo servir de base para comparação com a linguagem a ser proposta no capítulo IV deste trabalho. A

descrição a ser apresentada não pretende ser exaustiva, mas sim identificar as principais características relacionadas aos aspectos tempo real destas linguagens, enfatizando sempre que possível os requisitos e características introduzidos na seção anterior.

II.4.3.1 - ADA95

Características gerais - O novo padrão emergente para ADA, denominado inicialmente ADA9X e posteriormente ADA95, é visto como uma tentativa de endereçar os requisitos de aplicações específicas em uma linguagem de uso geral [Stoyenko 94][Burns 96a, 96b]. ADA95 é composta por um núcleo e por vários anexos (extensões), dentre os quais destacamos os anexos de orientação a objetos e de tempo real.

Com relação a tempo real, a principal característica introduzida em ADA95 a nível de núcleo, é a comunicação assíncrona entre tarefas, suportada diretamente pela construção "*protected type*" e pelo mecanismo ATC - Assynchronous Transfer of Control (conhecido como select assíncrono). O anexo tempo real é composto por uma série de pacotes que provêm, entre outras, as seguintes facilidades: controle dinâmico de prioridade, relógio de tempo real monotônico e capacidade para cálculo do tempo de CPU das tarefas.

Particularmente com relação a restrições temporais e previsibilidade, segundo [Stoyenko 94] e [Burns 96b], ADA95 introduz :

- O comando "*delay-until*", que faz com que uma tarefa espere por um tempo absoluto, permitindo a especificação de tarefas periódicas, podendo juntamente com o "select-assíncrono" ser usado como *timeout* das operações das tarefas, ou como forma de estabelecer um limite para o tempo de execução de um determinado segmento de código;

- Definição de tarefas periódicas e esporádicas através de *pragmas* (informações fornecidas ao compilador) que possibilitam a inferência do comportamento temporal de uma tarefa a partir de sua estrutura sintática;

- Maior liberdade no desenvolvimento de modelos de escalonamento, substituindo o modelo de prioridades de ADA por um conjunto padrão de opções baseado em escalonamento preemptivo por prioridades. O modelo de escalonamento a ser usado poderá ser selecionado pelo usuário via *pragmas* de configuração que permitem a definição da política de despacho ("*dispatching policy*"), política de enfileiramento de entradas ("*entry queuing policy*") e a política de bloqueio ("*locking policy*").

Com relação a análise de escalonabilidade, apesar das inovações introduzidas, claramente ADA95 continua permitindo que programadores escrevam programas cuja escalonabilidade não pode ser analisada, já que muitos aspectos, tal como uso de construções com tempo de execução ilimitado/imprevisível, não foram alterados no novo padrão proposto. Entretanto, segundo Stoyenko, em [Stoyenko 94], existem disciplinas de programação que podem ser impostas para tornar análise de escalonabilidade possível, e ADA95 provê características que viabilizam a definição de tal disciplina.

Avaliação da linguagem - Destacam-se em ADA95 a robustez, derivada de sua semântica precisa de tipos, e a sua flexibilidade relativa a configuração da política de escalonamento. Por outro lado, a representação de restrições temporais de forma indireta, não só dificulta o gerenciamento da complexidade como também o reuso e a manutenção dos STR produzidos; além disso, a análise de escalonabilidade só é possível através de uma disciplina de programação extremamente rígida, o que é problemático na medida em que ADA95 destina-se a programação de STR *hard*.

II.4.3.2 - Real-Time C++ (RTC++)

Características gerais - Desenvolvida na Carnegie Mellon University, RTC++ [Ishikawa 90, 92] é uma extensão de C++ para programação de aplicações tempo real, implementada sobre o kernel ARTS [Tokuda 89]. RTC++ foi projetada com base em um modelo de objetos ativos com restrições temporais, denominados objetos de tempo real, que interagem através de passagem de mensagens e sincronizam-se através de regiões críticas.

Restrições temporais - Em adição as classes de objetos C++, RTC++ introduz classes de objetos ativos ("*active class*"), que diferem das classes de objetos convencionais por apresentarem uma seção "*activity*" (local onde são definidas as *threads* correspondentes aos métodos do objeto) e por permitirem que a declaração de métodos seja estendida com a especificação de *deadline*, tempo de execução e manipulador de exceções temporais; tarefas periódicas podem ser definidas através da associação da cláusula "*cycle*" a um método. RTC++ também suporta especificação de restrições temporais a nível de comando, através das construções temporais "*within*", "*at*", "*before*" e "*cycle*".

Concorrência e sincronização - Concorrência em RTC++ é expressa e controlada através da definição de regiões criticas implementadas pelo comando "*region*"; sincronização condicional, por sua vez, é especificada através da associação de uma expressão guarda (cláusula "*when*") aos métodos dos objetos, a qual define as condições para ativação do método.

Herança - O mecanismo de herança também pode ser empregado na definição de objetos ativos; neste caso além de herdar as variáveis de instância e funções membro da classe "pai", uma classe "filho" pode também herdar a definição de *thread's* da seção "*activity*", as quais são consistentemente compostas com as novas definições da classe "filho".

Tratamento de exceções - Em RTC++, além da possibilidade de se associar funções destinadas ao tratamento de exceções relativas aos métodos de um objeto ativo, também é possível a associação de manipuladores de exceção (blocos iniciados com a palavra "*except*"), a quaisquer construções, temporais ou não, da linguagem.

Análise de escalonabilidade e suporte - Segundo [Stoyenko 92], RTC++ tem provisões suficientes para permitir análise de escalonabilidade e satisfaz a maioria dos requisitos de uma LTR. Em [Ishikawa 92], apresenta-se uma forma para estruturação das aplicações considerando os objetos ativos introduzidos e define-se um método para realização de análise de escalonabilidade de programa RTC++ assim estruturados. Quanto ao escalonamento, RTC++ utiliza a abordagem "*rate monotonic*".

Avaliação da linguagem - Destaca-se em RTC++, a simplicidade com que as restrições temporais são expressas (associadas diretamente aos métodos dos objetos) e o fornecimento de garantias temporais, viabilizando a realização de análise de escalonabilidade estática. Por outro lado, a especificação de restrições temporais a nível de comando e o mecanismo de concorrência utilizado dificultam o gerenciamento da complexidade e comprometem a capacidade de reuso e manutenção da linguagem. Além disso, a dependência de um ambiente operacional específico, embora possibilite o fornecimento de garantias, impede a utilização da linguagem em ambientes de propósito geral.

II.4.3.3 - DROL

Características gerais - Desenvolvida na Keio University (Japão), Drol [Takashio 92] é uma LTR-OO projetada como uma extensão de C++, fundamentada no modelo DRO e

implementada sobre o kernel ARTS [Tokuda 89]. DROL suporta a programação de STR distribuídos, oferecendo facilidades para : expressão de restrições temporais, invocação polimórfica temporal, detecção e manipulação de exceções temporais, definição da semântica de comunicação (através de protocolos especializados) e separação parcial dos aspectos funcionais dos aspectos temporais e de sincronização (através da filosofia de meta-objetos).

Objetos de tempo real - Adicionalmente aos objetos convencionais C++, DROL introduz objetos de tempo real, os quais são definidos (compostos) por três tipos de objetos : Objetos Básicos, Objetos *AbstractMeta* (ASM) e Objetos *ProtocolMeta* (PMETA), sendo os dois últimos definidos em um meta nível.

Objetos básicos implementam as funcionalidades da aplicação e possuem restrições temporais associadas a seus métodos. Objetos ASM, especificam o comportamento do objeto básico correspondente com relação a sincronização e gerenciam o atendimento de mensagens; estes objetos são compostos por uma parte base, que estabelece a correspondência com o objeto básico, e uma parte "*state*", que define os estados e as ações relativas aos métodos passíveis de serem executados em cada estado. O terceiro tipo de objeto (PMETA), é responsável pela definição de protocolos específicos (os quais dão semântica à comunicação inter-objetos) e pela criação e controle de objetos PWORKER, os quais são criados a cada ativação de um método e são responsáveis pelo controle das restrições temporais associadas aos métodos dos objetos básicos.

Comunicação - Além da possibilidade de definição de protocolos de comunicação por parte do usuário (via PMETA), DROL oferece três protocolos pré-definidos que são : síncrono, síncrono temporizado e assíncrono temporizado.

Uma das principais características de DROL é o mecanismo de invocação polimórfica temporal (realizado através dos comandos "*invoke*" e "*receive*"), o qual permite que um determinado método (dentre uma série de possibilidades) seja selecionado dinamicamente para execução, dependendo da disponibilidade de tempo no momento da invocação; Invocação polimórfica temporal juntamente com o conceito de objetos de tempo real, permitem a realização das propriedades de "melhor esforço" e "menor prejuízo" que nortearam o projeto do modelo DRO.

Sincronização - Em DROL, a sincronização é obtida através de um mecanismo de estados habilitados (similar aos estados habilitados, propostos em [Tomlinson 89]). Este mecanismo consiste na identificação dos possíveis estados em que um objeto pode se encontrar e na especificação dos métodos que podem ser ativados em cada estado. Com este mecanismo, DROL permite a separação total entre os aspectos funcionais e de sincronização de uma aplicação.

Restrições temporais - Em DROL, restrições temporais podem ser associadas tanto aos métodos quanto aos comandos de um objeto de tempo real. Todo método de um objeto básico deverá possuir um tempo máximo de execução (pior caso) associado a ele; além disso, Drol também permite a definição de métodos periódicos (ditos ativos), aos quais são associados tempo de início e de fim, período e *deadline*. A nível de comandos, DROL permite a associação das mesmas restrições temporais permitidas em RTC++.

Tratamento de exceções - Em DROL, funções destinadas ao tratamento de exceções relativas a violação das restrições temporais de um método podem ser associadas aos métodos (periódicos ou não) de um objeto básico. Além disto, DROL também permite que manipuladores de exceção sejam associados a uma invocação polimórfica, possibilitando que

exceções decorrentes da rejeição, da terminação anormal ou de um *timeout* relativos a invocação realizada, sejam detectadas e tratadas.

Outras características - A possibilidade de uso, bem como o impacto das características próprias do paradigma de objetos presentes em C++ (tais como herança e ligação dinâmica) sobre DROL, não foram abordadas em [Takashio 92, 93] e [Tokoro 93]; da mesma forma, nenhuma alusão é feita sobre a questão de análise de escalonabilidade.

Avaliação da linguagem - As principais vantagens de DROL são sua expressividade e sua flexibilidade, decorrentes da separação das questões temporais, de comunicação e de sincronização das funcionalidades da aplicação, possibilitada pelo uso de meta-objetos. Esta separação não só facilita o gerenciamento da complexidade, como também incrementa a capacidade de reuso e de manutenção dos programas produzidos. Também deve ser destacado em DROL, sua capacidade para definição de protocolos tempo real e a flexibilidade provida pelo mecanismo de invocação polimórfica.

Por outro lado, DROL não explora o potencial da reflexão para definição e controle de novos tipos de restrição temporal a nível de aplicação, apresentando um conjunto fixo de restrições temporais controlados pelo seu suporte de execução, o qual é dependente de um ambiente operacional específico; da mesma forma o controle do escalonamento também não pode ser realizado a nível da aplicação. Outro aspecto discutível em DROL é a existência de restrições temporais a nível de comando; aspecto este que não parece adequado à filosofia reflexiva e que pode comprometer as vantagens advindas desta filosofia.

II.4.3.4 - RealTimeTalk (RTT)

Características gerais - Desenvolvida na Suécia (Royal Institute of Technology), RTT se propõe a ser um sistema completo para desenvolvimento de STR *hard*, onde análise, projeto e implementação estejam fortemente relacionados e sejam baseados em conceitos e ferramentas que permitam que cada estágio do processo de desenvolvimento seja visto como um mero refinamento do estágio anterior [Gustafsson 94] [Eriksson 94]. Esta linguagem segue o modelo de programação RTT descrito na seção anterior.

A linguagem de programação RTT é baseada em Smalltalk, à qual são introduzidas algumas extensões tais como uso direto de métodos e comandos escritos em C e *loop's* limitados. Por outro lado, RTT apresenta algumas limitações, impostas a nível semântico, que coíbem a criação dinâmica de classes, o uso de recursão e de *loops* ilimitados. Adicionalmente, RTT implementa características tais como polimorfismo (via ligação dinâmica) e coletor de lixo ("*garbage collection*") através de técnicas deterministas, eliminando assim os principais focos de imprevisibilidade referentes ao paradigma de objetos.

A linguagem RTT não provê meios para explicitar restrições e exceções temporais a nível de linguagem; isto é feito no estágio de projeto da aplicação. Esta abordagem permite uma clara separação entre os aspectos funcionais de uma aplicação e seus aspectos temporais e de sincronização (especificados via grafos de precedência), não existindo nenhum mecanismo específico (a nível de linguagem) para expressão e controle de restrições temporais, concorrência e sincronização.

Análise de escalonabilidade - Através das limitações impostas a nível de linguagem e do uso de técnicas deterministas na implementação de seus mecanismos, RTT possibilita o cálculo automático do tempo de execução de seus programas, permitindo que a análise de escalonabilidade dos mesmos possa ser realizada.

Suporte de execução - O suporte de execução de RTT foi projetado para eliminar todas as características que pudessem causar não determinismo. As principais características deste suporte são:

- uso de uma técnica determinista (MTWC - Modified Two-Way Colouring table method [Eriksson 94]) para pesquisa de métodos;

- criação de objetos realizada com tempo de execução fixo;

- coletor de lixo automático com comportamento determinista [Gustafsson 94];

- interrupção de relógio como único tipo de interrupção permitido;

- sincronização realizada com base no tempo (não é utilizado nenhum mecanismo de sincronização dinâmico).

Ambiente de execução - Aplicações RTT são desenvolvidas em um ambiente Smalltalk/RTT e convertidas para um módulo de carga para o ambiente alvo onde serão executadas. Além do compilador RTT, o ambiente é composto por uma ferramenta destinada ao cálculo do tempo máximo de execução e por um escalonador "*off-line*" (baseado em pesquisa heurística). RTT não estabelece nenhum *hardware* ou sistema operacional específico para execução de suas aplicações, entretanto sua viabilidade depende de um ambiente (*hardware* e S.O.) previsível [Eriksson 94].

Avaliação da linguagem - Uma das principais contribuições de RTT é demonstrar que mecanismos típicos de linguagens orientadas a objetos, tais como ligação dinâmica e coleta de lixo automática, podem ser implementados de forma determinista; estes aspectos mais a proibição do uso de construções com tempo de execução ilimitado (tais como recursões e *loops* ilimitados), permite que o tempo máximo de execução (*worstcase*) dos programas RTT possam ser calculados e que estes programas tenham sua escalonabilidade analisada *off-line*.

Por outro lado, as limitações introduzidas reduzem a flexibilidade característica de SmallTalk e dificultam a programação de aplicações tempo real complexas. Além disso, a abordagem estática que caracteriza o modelo básico de programação RTT, embora necessária no contexto tempo real *hard*, não se mostra adequada aos requisitos de flexibilidade e composição com outros sistemas tempo real e mesmo com sistemas computacionais convencionais. Um aspecto questionável em RTT é o uso de uma linguagem extremamente dinâmica como Smalltalk, para implementação de um modelo de programação extremamente estático (segundo seus autores, a principal diferença entre RTT e MARS [Kopetz 91, 93], um ambiente voltado para sistemas tempo real estáticos, é que MARS baseia-se em processos enquanto RTT baseia-se em objetos). A questão, portanto, é saber se o uso de Smalltalk justifica-se, uma vez que muitas de suas potencialidades não serão utilizadas.

II.4.3.5 - FLEX

Características gerais - Desenvolvida na Universidade de Illinois [Lin 88] [Kenny 91] [Lin 91] como uma extensão de C++, FLEX é uma linguagem modular baseada no paradigma de computação imprecisa. Um programa FLEX, dependendo da disponibilidade de tempo e de recursos, pode produzir resultados mais ou menos precisos, porém sempre aceitáveis.

A abordagem adotada por FLEX é diferente das demais linguagens na forma como ela tenta garantir a satisfação das restrições temporais e de recursos. O tempo de execução dos

blocos com restrições pode ser ajustado dinamicamente através da substituição (progressiva e monotônica) de computações previstas originalmente, por alternativas menos precisas.

Restrições temporais - Restrições temporais e de recursos são expressas através de Blocos-Restrição - BR - ("*constraints blocks*"), os quais são constituídos por uma sequência de comandos precedida por uma lista de restrições, as quais podem ser lógicas, temporais ou de recursos (memória, por exemplo).

Restrições temporais podem ser expressas através dos seguintes atributos:

- "*start*" - tempo (absoluto) no qual a execução de um BR deve começar;
- "*finish*" - tempo (absoluto) no qual a execução de um BR deve terminar;
- "*duration*" - tempo de execução de um BR, deve ser menor ou igual a diferença entre "*finish*" e "*start*";
- "período" - tempo (relativo) entre as sucessivas execuções de um BR.

Todos os atributos usados para compor restrições temporais podem referir-se tanto ao tempo mais cedo ("*earliest time*", expresso pelo uso do operador >=) quanto ao tempo mais tarde ("*latest time*", expresso pelo uso do operador <=) relativo a ocorrência de um evento. Adicionalmente, na especificação das restrições de um determinado BR, é possível fazer-se referência aos atributos temporais de outros BR's.

O mecanismo de restrições temporais provido por FLEX, permite a definição de restrições totalmente dinâmicas e é expressivo o suficiente para explicitar as relações temporais mais importantes (especialmente as 13 relações aplicadas a pares de intervalos definidas por J.F. Allen [Kenny 91]).

Tratamento de exceções - Manipuladores de exceções podem, opcionalmente, serem associados aos blocos (seguindo a lista de restrições), os quais serão executados se alguma das restrições especificadas não for satisfeita, podendo recuperar, retomar ou terminar a execução do bloco onde ocorreu a exceção. O suporte da linguagem garante que, ou as restrições serão satisfeitas ou o BR em questão será "abortado" e o manipulador de exceções correspondente será executado.

Modelos de programação - Visando prover a correção funcional de seus programas FLEX dispõe de dois modelos de programação: computação imprecisa e polimorfismo de performance [Kenny 91].

Computação imprecisa, permite a consideração de resultados aproximados (parciais) quando a disponibilidade de tempo e de recursos não permite a obtenção de resultados exatos. Isto é implementado através da invocação de funções imprecisas (comandos "*impcall*" e "*impreturn*"), que permitem a definição das restrições temporais no momento da chamada (modelo síncrono) ou que de tempos em tempos retornem os resultados disponíveis (modelo assíncrono).

Polimorfismo de performance usa a idéia de múltiplas versões, ou seja, diversas implementações de uma mesma tarefa (as quais podem ser definidas como obrigatórias e opcionais); implementações estas que diferem na quantidade de tempo e recursos consumidos e na precisão dos resultados que elas produzem.

Escalonamento e análise de escalonabilidade - Ao contrário de outras LTR, FLEX não proíbe o uso de construções com tempo de execução ilimitado, além do que oferece uma abordagem muito dinâmica, dificultando a realização de análise de escalonabilidade

[Berryman 93]; entretanto, segundo Stoyenko, em [Stoyenko 92], com o uso de uma certa disciplina de programação é possível produzir programas analisáveis quanto a escalonabilidade. Quanto ao escalonamento, FLEX utiliza dois algoritmos : "*rate-monotonic*" (para tarefas obrigatórias) e "*earliest deadline first*" (para tarefas opcionais).

Avaliação da linguagem - Destaca-se em FLEX seu modelo de programação baseado em computação imprecisa, o qual introduz flexibilidade ao mesmo tempo em que procura garantir a satisfação das restrições temporais. Outro aspecto positivo de FLEX é a possibilidade de composição dinâmica de novos tipos de restrições temporais, aspecto este que torna FLEX mais expressiva que a maioria das linguagens aqui apresentadas; contudo, tais restrições são limitadas a combinação dos atributos previstos, sendo controladas pelo suporte de execução.

Por outro lado, o mecanismo usado para representação das restrições temporais, além de pouco legível, fere a uniformidade do modelo de objetos, dificulta o entendimento de programas e reduz a capacidade de reuso e manutenção da linguagem.

II.4.3.6 - Real-Time Java (RT-Java)

Características gerais - Real-Time Java [Nilsen 95, 96a] é um *superset* de Java 1.0 [SUN 95a, 95b, 95c] que introduz sintaxe adicional e semântica relacionada a tempo e memória para programas Java. Por outro lado, RT-Java é um *subset* de Java 1.0, na medida em que ela proíbe o uso de determinados aspectos de Java quando estes interferem na habilidade do sistema em suportar confiavelmente requisitos tempo real. Especificamente, RT-Java consiste de uma combinação de bibliotecas de classes especiais, protocolos padrão para comunicação com estas bibliotecas e a adição de duas estruturas de controle (*timed* e *atomic*) à sintaxe padrão de Java.

RT-Java objetiva o desenvolvimento de sistemas tempo real em geral e de sistemas embutidos em particular, sendo que um dos principais benefícios das extensões propostas é permitir a negociação e o gerenciamento confiável dos recursos de tempo e memória durante o desenvolvimento de aplicações tempo real. Embora RT-Java apresente características para representação e controle de aspectos temporais, a obtenção de um comportamento tempo real mais ou menos preciso depende da máquina virtual usada para execução de seus programas e também do ambiente (sistemas embutidos ou ambientes de propósito geral) no qual ela esta inserida. Máquinas Virtuais Java (JVM) convencionais não estão aptas a garantir tal comportamento, provendo no máximo um comportamento aproximado ao desejado. Por outro lado, JVM estendidas adequadamente para tempo real (RTJVM) podem garantir comportamento tempo real, mesmo em ambientes de propósito geral [Nilsen 95, 96b].

Aspectos temporais - Além da introdução do comando *timed* (cuja função é estabelecer um tempo limite para a execução de um segmento de código) e do comando *atomic* (cuja função é especificar que um segmento de código deve ser executado atomicamente), RT-Java introduz um pacote tempo real (Real-Time API) o qual estabelece um modelo de programação que permite a representação e o controle dos aspectos temporais de aplicações tempo real típicas. O componente central deste modelo é um executivo tempo real (uma das classes que compõem o RT-API), cuja função básica é tomar e garantir decisões relativas a alocação de recursos, incluindo-se serviços relativos a alocação de memória e escalonamento de tarefas.

Segundo o RT-API introduzido, uma aplicação (programa) tempo real deve ser organizado na forma de uma ou mais atividades tempo real (Real-Time *Activity*). Cada

atividade tempo real consiste de um número arbitrário de tarefas tempo real, um método para configuração (*configure()*) e outro para negociação (*negociate()*) dos recursos requeridos pela atividade. Uma atividade tempo real só é introduzida na carga de trabalho do sistema se houver disponibilidade de tempo e memória para sua execução, sem prejuízo das atividades aceitas anteriormente; os recursos necessários (para todas as tarefas que compõem a atividade) são especificados pelo método *configure()* e negociados com o *executivo* tempo real através do método *negociate()*.

Diferentes tipos de tarefas (*Periodic, Sporadic, Spontaneous e Ongoing*) tempo real são suportadas pelo RT-API, o qual provê serviços para análise e gerenciamento dos recursos de memória e CPU das diferentes tarefas que compõem uma atividade. Toda tarefa tempo real (independente de seu tipo) é composta por partes essenciais (métodos *startup()* e *finaliza()*) destinadas a expressar a funcionalidade mínima da tarefa e por um componente opcional (método *work()*) destinado a expressar a funcionalidade complementar da tarefa; sendo que *startup()* e *finalize()* devem ser analisáveis quanto ao seu tempo de execução.

Escalonamento e análise de escalonabilidade - O escalonamento das tarefas tempo real é realizado de forma on-line pelo *executivo* de tempo real, o qual usa diferentes estratégias (algoritmos de escalonamento) para despachar os diferentes tipos de tarefas tempo real. Segundo o modelo proposto, tarefas tempo real possuem maior prioridade que tarefas não tempo real para acesso aos recursos do sistema.

A análise de escalonabilidade é realizada implicitamente durante o processo de negociação realizado cada vez que uma nova atividade tempo real é introduzida no sistema; uma vez aceita para execução, uma atividade terá garantido recursos de memória e processamento para a execução de todas as suas tarefas. Entretanto, esta análise só é precisa através do uso de RTJVM com suporte para o cálculo do tempo de execução (*worstcase*) das partes essenciais e para medição das partes opcionais destas tarefas. Para possibilitar o cálculo do tempo de execução das partes essenciais, Real-Time Java estabelece uma série de restrições com relação as construções convencionais Java que podem ser utilizadas; entretanto, faz parte da filosofia de RT-Java reduzir ao mínimo os segmentos de código que precisam ter seu tempo de execução calculado, baseando-se sempre que possível nos tempos médios (que embora não permitam garantias, geralmente são mais adequados que o *worstcase* [Nilsen 95, 96a]).

Implementação - A proposta inicial de K. Nilsen consistia em implementar um pré-processador que convertesse código RT-Java utilizando o RT-API descrito em [Nilsen 96a], para programas Java convencional. Tais programas poderiam ser executados tanto em JVM convencionais quanto em máquinas virtuais Java especialmente projetadas para suportar as extensões tempo real introduzidas (RTJVM). Embora possível, o uso de JVM convencionais resultaria no máximo em um comportamento aproximado, enquanto o uso de RTJVM poderia garantir o comportamento temporal desejado.

Mais recentemente, K. Nilsen tem redirecionado seu trabalho para o desenvolvimento de PERC (Portable Executive for Reliable Control), um produto NewMonics Inc., que é ao mesmo tempo uma linguagem de programação e uma definição de máquina virtual. Como uma linguagem de programação, PERC é a própria linguagem Real-Time Java. Programas escritos em PERC podem ser traduzidos diretamente para Java através de um pré-processador (p2jpp na figura 2.2, extraída de [Nilsen 96c]) específico ou então podem ser traduzidos para *bytecode* Java com anotações usando o produto *Percolator*. Embora o código anotado possa ser executado por JVM convencionais (com suporte para o RT-API), só é possível garantir o

comportamento tempo real desejado quando este código for executado na máquina virtual PERC (que é uma RTJVM).

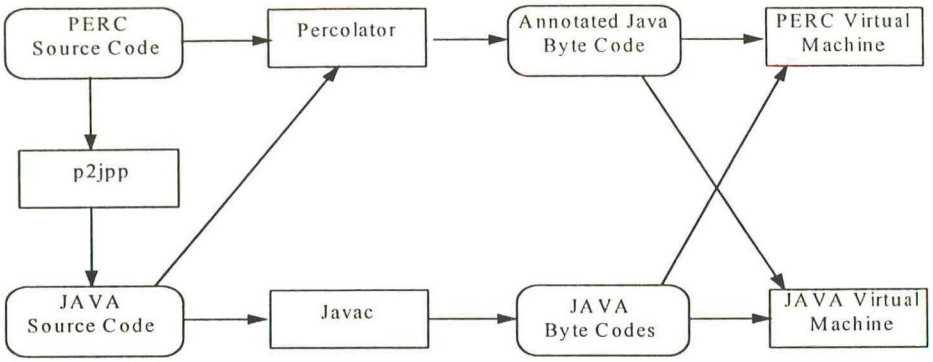


Figura 2.2 - Relacionamento entre PERC e JAVA

A máquina virtual PERC, além de ser totalmente compatível com Java e suportar integralmente o RT-API, implementa um coletor de lixo de tempo real (isto é, com comportamento temporal previsível) o qual evita que a aplicação seja interrompida arbitrária e imprevisivelmente como acontece com o coletor de lixo usado nas JVM convencionais. Além disso, a máquina virtual PERC está apta a entender as anotações presentes no *bytecode* gerado por *Percolator*, o que possibilita a análise do tempo de execução (*worstcase* e caso médio) do código gerado, e também a realização de transformações que fazem com que o código gerado seja mais previsível e eficiente.

Avaliação da linguagem - Destaca-se em RT-Java a possibilidade de se controlar recursos de tempo de CPU e memória diretamente a nível da aplicação. Outro aspecto positivo de RT-Java é a realização de análise de escalonabilidade na fase de negociação para admissão de uma atividade na carga de trabalho do sistema. A existência de um coletor de lixo automático determinista é outra contribuição importante da linguagem.

A existência de um conjunto fixo de tipos de tarefas (representando tipos de restrições temporais) e a necessidade de uma disciplina de programação rígida para enquadrar uma aplicação aos tipos de tarefas disponíveis, são as principais limitações da linguagem; isto, juntamente com as limitações impostas para obtenção de um comportamento previsível, dificultam a programação e comprometem a capacidade de reuso e manutenção características da linguagem Java.

II.4.4 - Comentários gerais sobre as linguagens apresentadas

Nesta subseção apresentaremos alguns comentários relativos aos requisitos e características básicas das linguagens apresentadas, com o objetivo de sintetizar o estado atual da área e justificar a proposição de uma nova LTR fundamentada nos paradigmas de orientação a objetos e reflexão computacional.

Análise de escalonabilidade - Na grande maioria das linguagens apresentadas análise de escalonabilidade só é possível através da proibição ou limitação do uso de mecanismos dinâmicos e de construções com tempo de execução ilimitado ou pelo menos não previsível. Contudo, como a imposição destas limitações (via especificação sintática e semântica) reduz a expressividade e a flexibilidade das linguagens, dificultando a programação de aplicações

mais complexas, a maioria das LTROO (RTT é uma exceção), especialmente aquelas voltadas para programação de STR *soft*, tem preferido considerar estas limitações na forma de disciplinas de programação, sob a responsabilidade do programador da aplicação.

Restrições temporais - Com relação a representação de restrições temporais, destacamos os seguintes aspectos:

- a representação explícita de restrições temporais, de forma simples e legível, contribui para o entendimento dos programas e conseqüentemente facilita o gerenciamento da complexidade e a manutenção dos sistemas;

- a associação de restrições temporais aos métodos dos objetos, além de manter a uniformidade do modelo de objetos, facilita o entendimento e favorece a reutilização do software;

- a associação de manipuladores de exceções temporais às construções temporais da linguagem, é fundamental para a segurança do sistemas;

- a possibilidade de se definir e controlar restrições temporais a nível de aplicação (via reflexão computacional, por exemplo), incrementa a flexibilidade e a adaptabilidade da linguagem a diferentes classes de aplicação, além de reduzir sua dependência de ambientes operacionais específicos.

Reuso, manutenção e facilidade de gerenciamento - Embora o uso de objetos favoreça a satisfação destes requisitos, a consideração de aspectos temporais pode comprometê-los; entretanto, este comprometimento pode ser minimizado se os aspectos relativos a representação de restrições temporais acima especificados forem observados. Além disso, a separação entre questões funcionais e de controle, possível através da abordagem de meta-objetos, também favorece a satisfação destes requisitos.

Independência de ambiente operacional - A forte dependência de ambientes operacionais específicos encontrada na maioria das LTR existentes, parece ter influenciado significativamente a pouca difusão destas linguagens e, por extensão, da tecnologia tempo real. Embora no caso de STR *hard* esta dependência justifique-se pela falta de características tempo real da maioria dos ambientes de propósito geral, no caso de STR *soft* as limitações destes ambientes podem ser contornadas através do suporte de execução das linguagens ou da transferência do controle dos aspectos temporais para a própria aplicação (via reflexão computacional, por exemplo), de forma a prover no mínimo um comportamento "*best-effort*".

Flexibilidade X previsibilidade - Para satisfazer o requisito de previsibilidade, via de regra, as linguagens tempo real tem sacrificado significativamente sua expressividade e flexibilidade, através da limitação dos mecanismos e construções que podem ser utilizados. Se por um lado esta postura satisfaz as necessidades básicas de muitos STR, por outro ela se mostra inadequada frente as novas exigências, tais como flexibilidade e capacidade de integração e de evolução, que cada vez mais caracterizam os atuais e futuros STR. Desta forma, torna-se evidente a necessidade de novas filosofias e novos mecanismos de programação tempo real, capazes de conciliar adequadamente o requisito de previsibilidade com o desejo de flexibilidade. Neste sentido, o uso conjunto de orientação a objetos e reflexão computacional apresenta-se como uma boa alternativa a ser explorada.

II.5 - Análise do tempo de execução

II.5.1 - Introdução

A análise de escalonabilidade de sistemas de tempo real depende do conhecimento do tempo máximo de execução (TME) das tarefas que compõem o sistema, o qual deve ser medido ou calculado previamente, conforme constatado na literatura sobre escalonamento e análise de escalonabilidade de sistemas de tempo real (a começar pelo clássico artigo de Liu e Layland [Liu 73], onde o TME das tarefas é assumido como sendo conhecido).

Além do conhecimento do TME das tarefas, vários outros fatores devem ser considerados para que a análise de escalonabilidade de um sistema tempo real possa ser realizada. Dentre estes fatores, podemos citar: modelo de execução, técnica de escalonamento utilizada, restrições temporais, restrições de sincronização, exclusão mútua, compartilhamento de recursos e comunicação inter-processos.

Neste trabalho, entretanto, nos limitaremos a explorar as questões diretamente relacionadas com a obtenção do tempo de execução das tarefas via cálculo (*worstcase*) e/ou medição (caso médio). Assim sendo, esta seção tem como objetivo fazer uma revisão da literatura existente sobre abordagens para obtenção do tempo de execução e sobre a utilização destas abordagens em linguagens de programação tempo real. Embora este trabalho esteja voltado para linguagens tempo real orientadas a objetos, nesta seção levaremos em conta também as linguagens tempo real não orientadas a objetos, haja visto que os fundamentos básicos são os mesmos e que a questão da análise do tempo de execução tem sido pouco explorada no contexto de orientação a objetos.

II.5.2 - Métodos para obtenção do tempo de execução

O tempo de execução de uma tarefa ou de um programa pode ser obtido através de dois métodos básicos: medição, que é um método dinâmico; ou cálculo, que é um método estático. Adicionalmente, a combinação destes métodos também é possível [Gustafsson 94] [Nilsen 95, 96a].

II.5.2.1 - Medição do tempo de execução

A medição do TME de um programa pode ser realizada através de várias abordagens e consiste em, literalmente falando, "medir" o tempo gasto na execução de um segmento, uma tarefa ou um programa; medição esta que é realizada antes do sistema ser colocado em produção. Dentre as abordagens existentes, podemos destacar:

- medição direta - que consiste na utilização de um analisador lógico para medir o tempo de execução de um segmento de código.

- monitoração de software - que consiste na introdução de instruções (tais como leitura do relógio do sistema) no código objeto para obtenção de dados temporais;

- simulação - que consiste na utilização de um software simulando o processador alvo;

Apesar de possível e comumente utilizado na prática, o método de medição de TME apresenta várias dificuldades técnicas, resultando em muitas desvantagens com relação ao método de cálculo que será visto na próxima seção. Dentre estas desvantagens, podemos destacar:

- dificuldade, ou mesmo impossibilidade de se considerar todos os possíveis caminhos de execução de um programa (normalmente a medição é baseada em casos de testes que não garantem cobertura completa). Isto eqüivale a dizer que a medição não nos fornece o tempo máximo de execução (*worstcase*), mas sim um tempo médio, sendo responsabilidade do projetista (ou programador) da aplicação determinar os casos de teste necessários para que o tempo médio obtido seja aceitável;

- os tempos medidos são válidos somente para a versão corrente do programa, qualquer alteração no programa exige a realização de uma nova e completa medição;

- os tempos resultantes de uma medição são válidos somente para o hardware utilizado, não podendo ser facilmente aproveitado para outros processadores;

- normalmente, não permite uma correlação entre os tempos medidos e o código fonte correspondente, dificultando a identificação dos segmentos do programa que precisam ser alterados;

- o mecanismo de medição pode causar interferência nos tempos medidos, fazendo-se necessário o uso de técnicas destinadas à correção desta interferência;

- a medição só pode ser realizada quando o sistema estiver concluído ou próximo de sua conclusão, impedindo assim a realização de avaliações intermediárias durante o processo de desenvolvimento;

- o tempo e o esforço humano consumidos no processo de medição, especialmente na preparação dos casos de teste, normalmente são excessivamente grandes.

II.5.2.2 - Cálculo do tempo de execução

O cálculo do tempo de execução é baseado na análise do código fonte e/ou código *assembly* dos programas, permitindo assim que a correção temporal de um STR possa ser avaliada sem a necessidade de sua execução prévia. Esta análise pode ser manual, automática ou mista; entretanto a análise manual só é factível para programas simples e pequenos, não sendo, portanto, de interesse no contexto deste trabalho.

O cálculo sistemático do TME, apesar de ser mais complexo, apresenta uma série de vantagens com relação ao método de medição; vantagens estas que o tornam bastante atrativo e alvo de inúmeros trabalhos de pesquisa. [Puschner 89], [Park 91, 93], [Vrchoticky 94], [Gustafsson 94] e [Harnon 94], entre outros, são trabalhos recentes que exploram a problemática relativa ao cálculo sistemático do TME, propondo métodos e ferramentas para sua solução.

Dentre as vantagens decorrentes desta abordagem, podemos destacar que ela:

- possibilita a realização de análise de escalonabilidade em tempo de compilação;

- considera todos os caminhos de execução possíveis, sendo portanto exaustiva;

- evita o envolvimento do programador no processo de re-cálculo decorrente de alterações no programa fonte;

- é mais amena a mudança de ambiente operacional, implicando (em alguns casos) apenas no fornecimento dos novos tempos das instruções do novo processador;

- permite a correlação entre o tempo de execução calculado e segmentos do programa fonte, facilitando a realização de modificações;

- permite a avaliação do comportamento temporal de partes do sistema ainda durante o seu desenvolvimento;

- reduz o tempo e esforço consumidos no processo de análise.

Fatores que influenciam no cálculo do TME - O cálculo do TME de um programa depende de muitos fatores, dentre os quais podemos destacar:

- código fonte do programa;
- linguagem e compilador utilizados;
- suporte de execução e sistema operacional;
- hardware envolvido (processador e outros dispositivos).

• O código do programa define o grafo do fluxo de controle e as instruções que serão executadas nos diferentes caminhos de execução. A análise deste código depende do tipo das construções utilizadas e de como estas instruções serão traduzidas para código *assembly* ou de máquina, sendo portanto fundamental o conhecimento da sintaxe e da semântica da linguagem de programação utilizada e das técnicas usadas pelo compilador nesta tradução. Limitações impostas na definição da linguagem (e implementadas por seu compilador) normalmente são necessárias para que determinadas construções de linguagem possam ser executadas dentro de um tempo limitado, possibilitando assim que esse tempo possa ser estaticamente calculado.

• O suporte de execução e o sistema operacional também afetam o tempo de execução através de fatores tais como: estratégia de execução utilizada, chamadas a serviços do sistema, mecanismo de comunicação utilizado etc. Assim sendo, suporte e sistema operacional devem garantir que estes fatores tenham tempo de execução previsível, para que o cálculo do TME do programa fonte possa ser realizado. Particularmente, o suporte de execução da linguagem deve ser capaz de gerenciar a possível falta de previsibilidade do sistema operacional e do hardware subjacentes, de modo a manter a previsibilidade oferecida a nível de linguagem.

• O hardware envolvido (especialmente o processador) constitui a base sobre a qual o TME será calculado, devendo portanto apresentar um comportamento determinista, e previamente conhecido. Otimizações de hardware tais como "cache", "pipelines" e "DMA", são fontes de problema para o cálculo do TME, uma vez que elas, apesar de otimizarem o comportamento médio do sistema, dificultam o cálculo dos tempos máximos (piores casos). Em [Gustafsson 94] são citadas algumas formas para solução destes problemas, que vão desde o uso de uma abordagem defensiva, evitando o uso de hardware otimizado, até o desenvolvimento de novos processadores (específicos para tempo real), passando por abordagens intermediárias nas quais tenta-se considerar as otimizações do hardware no cálculo do TME. Além disso, vários trabalhos têm sido propostos ([Zhang 93], e [Nilsen 95a] por exemplo) visando reduzir os efeitos do uso de hardware otimizado no cálculo do TME.

Pré-condições para o cálculo do TME - Segundo [Puschner 89], o principal problema relativo ao cálculo do TME é que o fluxo de controle de um programa em tempo de execução depende dos dados de entrada e do estado corrente das variáveis utilizadas; ou seja, os valores das variáveis usadas no controle de *loop's* e comandos seletivos e os valores dos apontadores de função determinarão o fluxo de execução do programa e conseqüentemente o seu tempo de execução.

Uma vez que é impossível calcular o tempo de execução de um programa considerando-se todos os possíveis valores que suas variáveis de controle podem assumir, algumas restrições (ou pré-condições) devem ser impostas para que o número de caminhos

alternativos (no grafo do fluxo de execução) seja finito e assim o TME possa ser calculado. As restrições tipicamente consideradas são as seguintes:

- o uso de *recursão* e de desvio incondicional deve ser proibido;

- apenas *loop's* limitados (por um número máximo de iterações ou por um tempo máximo de execução) podem ser utilizados;

- estruturas e mecanismos dinâmicos (coletor de lixo automático, por exemplo) não devem ser utilizados;

- a passagem de funções como parâmetros e o uso de ponteiros para funções deve ser proibido.

O afrouxamento das restrições acima, não necessariamente impede o cálculo do TME, mas certamente exige o uso de técnicas especiais (como é o caso por exemplo, do uso de coletores de lixo automático especialmente projetados para tempo real [Nilsen 96c]) e normalmente resultam no incremento da super-estimação dos tempos calculados (como é o caso, por exemplo, da solução usada em RTT [Gustafsson 94] para suportar polimorfismo).

A questão da super-estimação no cálculo do TME - Embora as pré-condições aqui apresentadas tornem possível o cálculo do TME de uma tarefa, em geral os tempos calculados desta forma são super-estimados, uma vez que todas as estimativas levam em conta sempre o pior caso. Esta super-estimação, apesar de levar a uma sub-utilização de recursos, é indispensável quando há necessidade de garantias, como é o caso de sistemas tempo real *hard.*

Gustafsson, em [Gustafsson 94], introduz o "Overreservation Factor" (OF), definido como sendo a razão entre o TME calculado e o tempo efetivamente gasto; fator este que é utilizado para expressar a qualidade do cálculo realizado.

Diversos são os fatores que contribuem para a super-estimação do TME calculado (aumento do OF), entre os quais podemos destacar:

- a existência de *loop's* interdependentes, onde é pouco provável que o pior caso de todos os *loop's* ocorra ao mesmo tempo;

- interdependência de seleções em um programa, onde as alternativas de pior caso nunca ocorrerão simultaneamente;

- uso de polimorfismo (via ligação dinâmica) em programas orientados a objetos, onde a única maneira de possibilitar o cálculo do TME estaticamente, é considerar sempre o método com maior TME entre todos os métodos cujo nome corresponde ao seletor utilizado na ativação em questão;

- uso de hardware otimizado (*cache, pipeline e DMA*).

Para minimizar o problema de super-estimação e conseqüentemente permitir que os tempos calculados estejam mais próximos dos tempos efetivamente gastos na execução, várias abordagens ([Puschner 89] e [Park 93], por exemplo) têm sido propostas na literatura, e serão comentadas na próxima seção.

II.5.3 - Visão geral das principais abordagens usadas para obtenção do TME

Nesta seção, visando exemplificar o estado da arte na área de predição do TME de programas, daremos uma visão geral de algumas das principais abordagens propostas para a solução deste problema que continua em aberto.

II.5.3.1 - Abordagem usada em RT-Euclid

RT-Euclid [Stoyenko 86, 91] é um ambiente para desenvolvimento de STR composto por compilador, analisador de escalonabilidade e sistema de execução. Em RT-Euclid o cálculo do TME dos processos é possível, já que todas as construções da linguagem possuem tempo de execução limitado; este cálculo é realizado pelo analisador de escalonabilidade.

O analisador de escalonabilidade é dividido em dois estágios:

- "Front-end" - é a parte do analisador dependente da linguagem (implementado como parte do compilador), cuja funcionalidade básica é o cálculo parcial do TME (pior caso) para os processos que compõem um programa;

- "Back-end" - é independente da linguagem e sua funcionalidade básica consiste em concluir o cálculo do TME dos processos e predizer, a partir das informações geradas no "front-end", se as restrições temporais especificadas podem ou não ser garantidas.

A estratégia usada pelo "front-end" consiste em dividir o código do programa em segmentos (simples ("straight-lines-of-code"), chamadas internas, externas e de sistema, comunicação e seletores), a partir dos quais é construída uma árvore de segmentos onde são registrados os tempos de execução (calculados a partir do código *assembly*) relativos aos segmentos simples e seletores. Neste estágio, o tempo de execução relativo aos segmentos de comunicação, chamadas (internas, externas e de sistema) e seletores (contendo segmento de comunicação ou chamadas) não é calculado; da mesma forma, nenhuma informação sobre o modelo de execução e sobre escalonamento é considerada neste estágio.

O primeiro passo do "back-end" é, a partir da árvore construída no estágio anterior, resolver as chamadas internas e externas (substituindo-as pelo corpo dos procedimentos correspondentes) e os seletores, determinando seus tempos de execução e registrando-os na árvore de segmentos. O próximo passo do "back-end" é verificar se as restrições temporais especificadas podem ou não ser garantidas, o que é feito considerando-se os tempos registrados na árvore de segmentos e os *delay's* de comunicação especificados no programa fonte (paralelamente, será determinado o TME relativo aos segmentos ainda não resolvidos). Esta verificação é realizada através da simulação do modelo de execução [Stoyenko 91].

II.5.3.2 - Abordagem usada no ambiente "MARS"

MARS (MAintainable Real-time Systems) [Kopetz 91, 93] é um ambiente completo para desenvolvimento de aplicações tempo real *hard*, incluindo arquitetura, sistemas operacional, metodologia de projeto e ambiente de programação.

A abordagem utilizada no ambiente MARS para análise do TME, baseia-se no cálculo do TME dos programas escritos em MODULA/R, que é uma linguagem de programação projetada especificamente para MARS e que contém apenas construções com tempo de execução limitado. O cálculo do TME é realizado com base em uma árvore temporal ("timing tree") construída a partir do código fonte dos programas. Nesta árvore são registrados os tempos de execução correspondentes ao código *assembly* destes programas e opcionalmente os tempos estimados pelo usuário, permitindo o cálculo do TME real ou hipotético.

O tempo de execução relativo a cada construção básica (primitivas, seqüências, seleções, loop's e procedimentos) é calculado com base em uma fórmula específica para cada construção, a qual descreve a forma como o tempo de execução deve ser calculado [Pospischil 92]. Uma vez calculado o TME das construções básicas, o analisador de tempo de execução

atravessa a árvore recursivamente, calculando o tempo de execução das demais construções. Em seguida, a árvore temporal é convertida para um conjunto de grafos conectados e direcionados (um grafo por procedure), onde as arestas representam um segmento de código e são valoradas pelo tempo de execução correspondente, e os vértices representam os pontos de "fork" e "join" do fluxo de controle. Depois disto, usando algoritmos de programação linear, é determinado o maior caminho do grafo. O somatório dos valores das arestas que compõem o maior caminho encontrado, corresponderá ao TME (pior caso) da procedure analisada.

Um aspecto interessante a ser ressaltado nesta abordagem, é o uso de construções especiais ("scope's, "marker's" e "loop's sequences" [Puschner 89] [Vrchoticky 94]), inseridas no código da aplicação. Através destas construções pode-se expressar dependências entre as unidades sintáticas e informações relativas a execução do algoritmo implementado. O uso destas construções é considerado nas fórmulas utilizadas para cálculo do TME, permitindo a redução da super-estimação dos tempos calculados.

Os resultados da análise de TME são usados no ambiente MARS para dois propósitos:

1 - Como "feed-back" para o programador, que, comparando os tempos calculados para cada segmento com os tempos esperados, poderá decidir pela necessidade ou não da alteração do código;

2 - Como base para a verificação da escalonabilidade das tarefas, levando em conta suas restrições temporais e de precedência, e as propriedades temporais do hardware e do software subjacentes.

II.5.3.3 - Abordagem usada no ambiente "RTT"

As idéias básicas da abordagem utilizada em RTT [Gustafsson 94] [Eriksson 94] para cálculo do TME, podem ser resumidas nos seguintes passos:

1- Decomposição do código fonte de um programa RTT em componentes básicos. Nesta etapa cada método produz uma função C contendo macros-C, as quais serão traduzidas para *assembly;*

2 - Tradução do código fonte C (usado na forma de funções "in-line") para *assembly;*

3 - Cálculo do tempo de execução do código *assembly* gerado nos passos anteriores;

4 - Cálculo do TME dos componentes básicos do programa RTT, a partir do tempo calculado para o código *assembly* correspondente a cada componente;

5 - Verificação da existência de recursões;

6 - Cálculo final dos tempos de execução do código RTT.

Para realização destas tarefas, o analisador é dividido em "front-end" e "back-end", sendo que o "front-end" é dependente da linguagem e sua funcionalidade básica é analisar o código a nível da linguagem RTT (a partir das informações temporais fornecidas pelo "back-end". O "back-end", por sua vez, é dependente do processador e sua função básica é analisar o código *assembly* gerado.

Para calcular o TME dos componentes básicos (seqüências, blocos, seleções e repetições) de um programa RTT (passo 4), o "front-end" utiliza uma série de fórmulas especificas para cada componente (e sua variações), baseadas no tempo de execução das macros-C que representam cada componente. No passo 6, o tempo relativo a ativação de

métodos é determinado e os tempos de execução (máximo e mínimo) dos métodos RTT são determinados e passados para o escalonador do sistema.

Outros aspectos da abordagem - O fato de que a linguagem RTT é uma linguagem orientada a objetos e polimórfica, dificulta a detecção de recursões, uma vez que em programas orientados a objetos existem 3 tipos diferentes de recursão: verdadeira, de classe e polimórfica. A solução adotada em RTT foi a proibição de todos os tipos de recursões, embora recursões de classe e polimórfica não impliquem em tempo de execução imprevisível.

Outro problema decorrente do fato que RTT é orientada a objetos, é a existência de polimorfismo (implicando em ligação dinâmica); a solução neste caso, visando permitir o cálculo do TME estaticamente, foi considerar sempre o pior caso; ou seja, na presença de métodos com mesmo nome em diferentes objetos, será sempre considerado aquele com maior tempo de execução, implicando assim na super-estimação dos tempos calculados. O tempo gasto na procura de um método não impossibilita o cálculo, uma vez que RTT utiliza uma técnica determinista de pesquisa (MTWC - Modified Two-way Coloring table method [Gustafsson 94]).

Visando reduzir a super-estimação dos tempos calculados, foram propostas duas novas construções: "execution passage counter" e "path checker". Estas construções serão usadas no código da aplicação, com o objetivo de permitir a detecção de caminhos não realizáveis no fluxo de execução.

Segundo Gustafsson, em [Gustafsson 94], a implementação de um protótipo da abordagem proposta, mostra que seus princípios básicos estão corretos; entretanto, alguns aspectos (incluindo "garbage collection") não foram levados em conta.

II.5.3.4 - Abordagem usada em Real-Time Java

Real-Time Java utiliza uma abordagem mista para obtenção do tempo de execução de seus programas: cálculo, para os componentes essenciais (métodos *startup()* e *finalize()* das tarefas tempo real e segmentos de códigos associados a cláusula *atomic*), e medição para os demais casos.

Ambos, cálculo e medição do tempo de execução são realizados através de métodos específicos (*analise()* e *cputime()*) presentes na classe *executive* do Real-Time API [Nilsen 96a] e são suportados pelo analisador de *bytecode* de máquinas virtuais Java especialmente implementadas para prover comportamento tempo real (PERC [Nilsen 96c], por exemplo).

Em RT-Java, a análise do tempo de execução dá-se durante a etapa de configuração das atividades de tempo real. Nesta etapa, o método *configure()* ativa os métodos *análise()* e *cputime()* (do *executivo* tempo real que esta gerenciando a aplicação) para obter, respectivamente, o tempo máximo de execução (*worstcase*) dos segmentos de código considerados essenciais (via cálculo), e do tempo médio de execução dos demais segmentos de código (via medição); observe-se que é responsabilidade do programador determinar o número de medições necessárias para obtenção de um tempo médio representativo.

O cálculo do tempo de execução de um segmento de código é realizado sobre o *bytecode* gerado a partir deste segmento, representado na forma de um grafo de fluxo de controle, não sendo possível qualquer acesso ao código fonte original; assim sendo o que é ou não analisável deve ser caracterizado formalmente a nível de *bytecode*. Informalmente, entretanto, para que um segmento de código fonte possa gerar *bytecode* analisável (e

conseqüentemente possa ter seu tempo máximo de execução calculado), ele deve ser composto apenas por:

- invocação de *métodos finais*, cuja implementação consiste inteiramente de código analisável;
- comandos *if-then-else* para os quais a expressão de controle e o código associados às cláusulas *then* e *else* sejam analisáveis;

analisáveis;

- comandos *for-loop*, onde as expressões de controle e o corpo do *loop* são analisáveis e a variável de controle possui limites (inferior e superior) constantes, é incrementada/decrementada por valores constantes e não é modificada no corpo do *loop*;
- comandos *timed*, cujo limite de tempo especificado é constante (neste caso, o tempo de execução será o próprio limite de tempo especificado, independentemente da complexidade do código associado).

II.5.3.5 - Abordagem baseada em esquemas temporais

A abordagem proposta por Shaw e Park [Park 93] [Gustafsson 94] na University of Washington, baseia-se na definição de esquemas temporais ("timing schema") para as construções básicas do programa fonte. Estes esquemas nada mais são do que fórmulas usadas para o cálculo dos tempos de execução máximo e mínimo destas construções.

A idéia básica desta abordagem consiste em : decompor os comandos do programa fonte em componentes básicos (por exemplo, o comando "while *<Expressão>* do *<Comando>*" é decomposto em *<expressão>* e *<comando>*); calcular o tempo de execução destes componentes (a partir do código objeto dos mesmos); e compor os tempos calculados de acordo com os esquemas temporais predefinidos. Aplicando decomposição, predição e composição recursivamente, o TME de comandos, sequências e funções é calculado.

Os experimentos realizados, usando um sub-set de C em uma máquina SUN, mostraram que o esquema temporal provê predições seguras e úteis no caso geral [Park 93]; entretanto, para programas mais complexos, onde a possibilidade de caminhos não realizáveis é maior, a predição é muito pobre, uma vez que estes caminhos também são considerados.

Visando melhorar a qualidade do cálculo efetuado, os autores propuseram uma extensão da abordagem ("path model" [Park 93]), na qual os caminhos não realizáveis podem ser detectados e eliminados a partir de informações sobre a execução fornecidas pelo usuário. Exemplos destas informações são: número de iterações de um *loop* e relações existentes entre comandos. Estas informações são expressas através de uma linguagem de descrição de informação (IDL - Information Description Language) e posteriormente traduzidas para expressões regulares que serão usadas para a detecção de caminhos não realizáveis no programa. Assim sendo, o esquema temporal proposto originalmente levará em consideração apenas os caminhos factíveis.

II.5.3.6 - Outras abordagens

Nesta subseção, consideraremos outras abordagens que merecem ser citadas e que deverão ser analisadas mais detalhadamente em trabalhos futuros.

Anotação de programas - Proposta por MOK na University of Texas [Mok 89] [Gustafsson 94], esta abordagem visa a especificação e a avaliação do tempo máximo de execução de programas escritos em C. A idéia básica da proposta consiste em "anotar" programas, isto é, marcar programas com informações extras que serão usadas no cálculo do TME. Durante a tradução do programa anotado (realizada por um compilador C modificado) o tempo de execução do código existente entre 2 marcadores é calculado através de simulação do hardware. Os tempos obtidos são usados para o cálculo do TME final, segundo um script TAL (Timing Analysis Language) gerado na compilação do programa anotado; este script pode ser modificado manualmente para melhorar a qualidade dos tempos calculados.

Abordagem usada em DEDOS - O analisador temporal de DEDOS segue uma abordagem analítica, baseada nos trabalhos de Park [Park 93] e Puschner [Puschner 89]. Basicamente a abordagem consiste em dividir o código assembly ou de máquina obtido da tradução de programas C++ em pequenos fragmentos, nos quais não existe dependência de dados. Em seguida o tempo de execução destes fragmentos é calculado (com base nos ciclos de *clock* de cada instrução) e estes tempos são combinados de acordo com as construções de alto nível que eles representam, através de esquemas temporais pré-definidos. Implicações decorrentes do uso de uma linguagem orientada a objetos (C++), não são reportadas na referência disponível [Gustafsson 94].

Avaliação parcial - A abordagem proposta por Nirkhe e Pugh [Nirkhe 93], denominada avaliação parcial, é uma técnica de transformação de programas na qual as informações disponíveis em tempo de compilação sobre o ambiente de execução são usadas para derivar um programa "residual" especializado para este ambiente. O programa residual obtido, além de mais eficiente, é extremamente mais simples que o programa original, podendo ter seu TME calculado mais fácil e precisamente. Segundo seus autores, a abordagem proposta não é uma alternativa a outras abordagens existentes mas sim uma abordagem complementar a elas. Um protótipo desta abordagem foi implementado sobre o sistema MARUTI [Nirkhe 93], para avaliação de programas escritos em uma linguagem experimental projetada especificamente para este fim.

Micro-análise - Proposta por Harnon et al. [Harnon 94], esta abordagem consiste em:

1 - compilar o programa e efetuar o "disassembler" do módulo objeto resultante;

2 - transformar as instruções de máquina em uma sequência de operações primitivas (micro-instruções) que expressem as funcionalidade destas instruções. Este processo é denominado micro-tradução;

3 - Analisar as operações primitivas (de acordo com os padrões pré-definidos na linguagem MDL - Machine Description Language), determinando seu tempo de execução (representado pelo número de ciclos necessários para sua execução). Quando a análise termina, o tempo de execução do segmento de código objeto analisado é computado e fornecido para o usuário na forma de um intervalo que especifica o melhor e o pior caso.

Esta abordagem foi implementada e tem sido usada para predizer o TME de programas C, ADA e Assembly, tomando como base os processadores Motorola MC68020 e Intel 80386. A implementação realizada oferece uma interface interativa através da qual o usuário pode fornecer as informações necessárias (segmento de código a ser analisado, número mínimo e máximo de iterações de um *loop*, etc.) para realização do cálculo do TME.

II.5.4 - Considerações finais relativas ao cálculo do TME

Na presente seção, foram apresentadas diversas abordagens que, de uma forma ou de outra endereçam a questão do cálculo do tempo de execução. A partir destas abordagens, entre outras coisas, é possível concluir que:

1 - Ainda não foi encontrada uma abordagem completamente satisfatória e abrangente relativa a predição estática do tempo de execução de programas, que desta forma, continua sendo uma área de pesquisa em aberto.

2 - A predição estática só é possível quando restrições relativas ao uso de construções imprevisíveis são impostas a linguagem, ou quando é sub-entendida a interferência do programador no processo de cálculo.

3 - É possível manter uma correlação entre os tempos calculados a partir do código assembly (ou alguma forma intermediária) e as construções de alto nível correspondentes, facilitando com isso o processo de desenvolvimento de programas tempo real.

4 - Além da imprevisibilidade, um dos principais problemas detectados é a super-estimação dos tempos calculados. Uma solução consensual para este problema, tem sido a introdução de informações sobre a execução (através de construções especiais, anotações, *pragmas*, etc.), visando por exemplo a eliminação de caminhos não realizáveis. Esta solução também contribui para a redução da complexidade do problema de cálculo do TME, que geralmente é NP-completo em função da possível explosão do número de caminhos do grafo do fluxo de execução, quando este grafo é derivado de uma análise realizada exclusivamente com base na estrutura do programa.

5 - Particularmente no caso de linguagens orientadas a objetos, questões como polimorfismo (ligação dinâmica), coleta de lixo automática e busca de métodos em hierarquias de classes, que normalmente tem sido consideradas problemáticas com relação a previsibilidade, têm encontrado soluções satisfatórias (veja por exemplo os casos de RTT e RT-Java). Por outro lado, aspectos mais avançados como por exemplo herança dinâmica e criação dinâmica de classes, não têm sido considerados.

6 - Apesar do progresso de pesquisas recentes, o uso de hardware otimizado continua implicando significativamente na super-estimação dos tempos de execução obtidos via cálculo. Soluções satisfatórias para esta questão constituem um campo aberto de pesquisa.

7 - Não foram encontradas referências relativas a análise do tempo de execução de programas escritos em linguagens com suporte a reflexão computacional.

II.6 - Conclusões

Este capítulo apresentou as potencialidades e limitações dos paradigmas de orientação a objetos e reflexão computacional para programação de sistemas tempo real e descreveu os principais modelos e linguagens de programação tempo real baseados nestes paradigmas. Com relação aos modelos e linguagens apresentadas, foram identificadas suas principais características, vantagens e limitações e também foi realizada uma breve avaliação dos mesmos. Adicionalmente, a questão da obtenção e da análise do tempo de execução de programas tempo real também foi discutida.

Com base no estudo realizado, constatamos que o uso de orientação a objetos e reflexão computacional, pode contribuir significativamente na solução de muitos dos problemas atualmente encontrados na programação de aplicações tempo real. Por outro lado, constatamos também que o potencial destes paradigmas não é completamente explorado em nenhum dos modelos e linguagens apresentados, especialmente se considerarmos os novos requisitos dos atuais e futuros sistemas tempo real, tais como flexibilidade, capacidade de integração e de evolução e independência de ambiente operacional.

Assim sendo, concluímos que há necessidade de novos modelos e linguagens que explorem mais profundamente o potencial dos paradigmas de orientação a objetos e reflexão computacional com o objetivo de melhor satisfazer os requisitos (especialmente os novos) dos sistemas tempo real. Neste sentido, este trabalho propõe um modelo e uma linguagem de programação baseados nestes paradigmas que, como será visto nos próximos capítulos, representa uma evolução relativa aos existentes, na medida em que reúne e amplia as vantagens neles dispersas e reduz as limitações existentes.

Capítulo III - O Modelo RTR

III.1 - Introdução

Os sistemas tempo real (STR) caracterizam-se hoje, cada vez mais, por um alto grau de complexidade, pela necessidade de flexibilidade, pelo requisito de integração entre partes tempo real e não tempo real e por necessitarem ser executados em ambientes diversificados. Para atender estas necessidades, propomos neste trabalho um modelo de programação tempo real, denominado Modelo RTR (Modelo Reflexivo Tempo Real), que permite que STR possam ser estruturados e programados de forma sistemática, confiável e flexível. Além de favorecer a obtenção de correção temporal (requisito básico de todo STR), o modelo RTR também tem como objetivos:

- facilitar o gerenciamento da complexidade de STR;
- flexibilizar a programação de STR, permitindo que as questões temporais possam ser expressas a nível computacional tão naturalmente quanto possível;
- ser independente de linguagem de programação e de ambiente operacional;
- favorecer o reuso e a manutenção do software desenvolvido;
- reduzir o "*gap*" semântico existente entre projeto e programação de STR;
- favorecer a extensão e a evolução dos sistemas, permitindo sua adaptação a mudanças internas (no sistema) e externas (no ambiente operacional);
- gerenciar os conflitos decorrentes da necessidade de previsibilidade com o desejo de flexibilidade.

Este capítulo está estruturado da seguinte forma: inicialmente é apresentada uma visão geral do modelo RTR a partir de suas características básicas, de sua estrutura geral e de sua dinâmica de funcionamento; em seguida apresenta-se a estrutura e a semântica informal dos diversos componentes do modelo, os quais são didaticamente exemplificados. Na seqüência, após a expressividade do modelo ser demonstrada através da identificação de várias situações para as quais o modelo mostra-se adequado, é apresentada e exemplificada uma extensão do modelo RTR para ambientes distribuídos abertos. Encerrando o capítulo, é realizada uma análise comparativa entre o modelo proposto e outros modelos existentes.

III.2 - Caracterização do Modelo RTR

O Modelo RTR é um modelo de programação para aplicações tempo real que se caracteriza como sendo uma extensão concorrente, reflexiva e tempo real do modelo de objetos convencional. No que segue, descreveremos a forma pela qual as características básicas do modelo podem contribuir para que seus objetivos sejam alcançados.

Orientação a objetos - Por ser uma extensão do modelo de objetos convencional, o modelo proposto herda seus mecanismos de estruturação e conseqüentemente sua potencialidade relativa ao entendimento, reuso, extensão e manutenção de sistemas. Complementarmente, por ser orientado a objetos, o modelo proposto possibilita um

tratamento adequado das questões de concorrência e distribuição inerentes a muitos dos STR atuais, além de facilitar o uso de reflexão computacional.

Reflexão computacional - Reflexão computacional é introduzida no modelo RTR através da abordagem de meta-objetos [Maes 87], que estabelece a separação dos aspectos funcionais da aplicação (implementados através de objetos-base) do controle de seu comportamento (implementado através de meta-objetos). O uso de reflexão computacional flexibiliza o desenvolvimento de STR, na medida em que permite que as políticas de controle sejam modificadas e/ou substituídas (inclusive dinamicamente), sem que os objetos base da aplicação e o suporte de execução necessitem ser modificados; assim sendo, a evolução do sistema bem como sua independência de ambiente operacional ficam facilitadas. No modelo proposto, todos os aspectos temporais (restrições, exceções e escalonamento) e mais os aspectos de concorrência e sincronização são tratados de forma reflexiva, possibilitando o uso de diferentes mecanismos e políticas no controle do comportamento dos aspectos refletidos. Adicionalmente, a separação explicita entre questões funcionais e não funcionais, contribui para o entendimento do sistema, favorece o reuso e a manutenção de objetos-base e meta-objetos e incrementa a produtividade do programador.

Tempo real - O modelo de objetos convencional não possui suporte para representação e controle dos aspectos temporais das aplicações, e portanto, deve ser estendido. No modelo proposto, esta capacidade é obtida a partir da representação explícita das restrições temporais a nível de objetos-base (as quais são associadas a declaração e/ou ativação de métodos) e a verificação e controle destas restrições a nível de meta-objetos. Outra questão fundamental relativa a obtenção de correção temporal é a presença de um escalonador tempo-real operando no meta-nível da aplicação, o qual pode ser escolhido pelo usuário.

O esquema reflexivo usado na representação/controle das questões temporais, flexibiliza a definição e o uso de restrições temporais e algoritmos de escalonamento aumentando a possibilidade de que as restrições temporais sejam satisfeitas. Restrições temporais e algoritmos de escalonamento podem ser escolhidos em função da especificidade de cada aplicação, de forma independente do suporte de execução. Contudo, em função de suas características e coerente com suas finalidades (modelagem e programação de STR *soft*), o modelo RTR segue uma abordagem dita de melhor esforço (*best-effort*), segundo a qual tenta-se evitar que as tarefas executem fora de suas especificações temporais e garante-se que toda violação temporal será detectada e que a ação alternativa (exceção temporal) correspondente a esta violação será executada.

Concorrência - O modelo RTR permite que a concorrência inerente ao mundo real possa ser mapeada diretamente a nível de sistema computacional, favorecendo a obtenção de correção temporal, na medida em que as restrições temporais associadas às diferentes tarefas do sistema podem ser consideradas simultaneamente antes de uma decisão de escalonamento. O modelo proposto permite apenas concorrência externa (isto é, concorrência entre objetos mas não entre métodos de um mesmo objeto) cujo controle (exclusão mútua na execução dos métodos de um objeto-base) é realizado de forma reflexiva a nível de meta-objetos; esta concorrência é provida pela existência de objetos ativos e pelo uso de mensagens (chamada de métodos) assíncronas. Outrossim, a sincronização condicional entre métodos de um objeto-base é tratada reflexivamente.

III.3 - Estrutura geral e dinâmica de funcionamento

Estrutura geral - Estruturalmente o modelo RTR é composto por objetos-base de tempo real, meta-objetos gerenciadores (um para cada objeto-base tempo real existente), um meta-objeto escalonador e um meta-objeto relógio, os quais interagem através de passagem de mensagens síncronas e assíncronas. A figura 3.1 esquematiza a estrutura geral do modelo RTR e o inter-relacionamento entre seus componentes básicos; nos próximos parágrafos, visando um melhor entendimento da estrutura do modelo, descrevemos rápida e superficialmente as funções básicas de cada um deles.

Os Objetos-Base de Tempo Real (OBTR) implementam a funcionalidade da aplicação e, em adição ao modelo de objetos convencional, podem ter restrições temporais e manipuladores de exceções temporais associados à declaração e ativação de métodos.

Os Meta-Objetos Gerenciadores (MOG) gerenciam o comportamento dos objetos-base, sendo responsáveis pelo gerenciamento das requisições para ativação dos métodos de seus objetos-base correspondentes, pelo controle de concorrência na execução desses métodos, pelo gerenciamento das restrições de sincronização e pelo processamento das restrições temporais e das exceções temporais associadas aos métodos dos objetos-base.

O Meta-Objeto Escalonador (MOE) tem como função básica receber, ordenar e liberar os pedidos de escalonamento (provenientes dos MOG) segundo uma determinada política de escalonamento, visando fornecer os parâmetros necessários ao escalonamento realizado pelo suporte subjacente.

O Meta-Objeto Relógio (MOR) tem como função controlar a passagem do tempo (visando a detecção de violação das restrições temporais) e realizar ativações programadas para um tempo futuro (ativações "*time-trigger*").

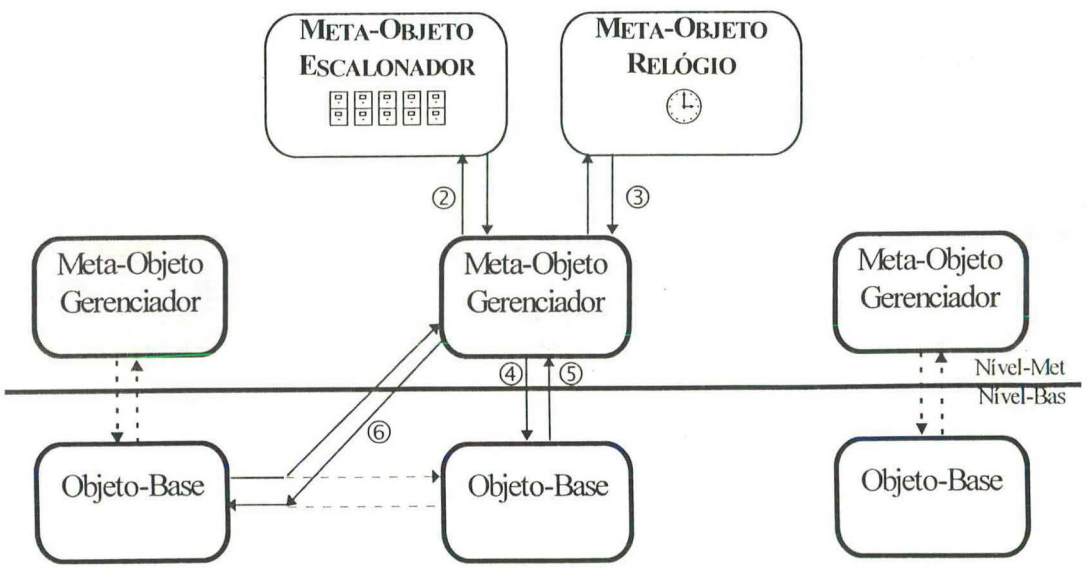


Figura 3.1 - Estrutura geral do modelo RTR

Dinâmica de funcionamento - A dinâmica do modelo RTR, envolvendo a interação entre objetos-base e seus meta-objetos, pode ser observada na figura 3.1. Um pedido de ativação de um método de um determinado objeto-base é desviado para seu meta-objeto

gerenciador correspondente (ação ① da figura 3.1). O meta-objeto gerenciador, por sua vez, interagirá com o meta-objeto escalonador (ação ②) e com o meta-objeto relógio (ação ③) para processar as restrições temporais associadas ao método solicitado, e se estas restrições não forem violadas, ativará o método solicitado no objeto-base (ação ④), retornando em seguida, via meta-objeto gerenciador para o objeto que deu origem a chamada (ações ⑤ e ⑥).

III.4 - Descrição detalhada

Nesta seção será apresentada uma descrição detalhada da estrutura e das funcionalidades dos diferentes componentes do modelo RTR (objetos-base e meta-objetos); adicionalmente, serão apresentados exemplos didáticos destes componentes. Estabeleceremos uma sintaxe concreta (representada através de uma pseudo linguagem) para descrever a estrutura do modelo e descreveremos a sua semântica informalmente, através da descrição das funcionalidades de seus componentes.

III.4.1 - Objetos-Base de Tempo Real (OBTR)

Estrutura geral - Os objetos-base de tempo-real, implementam as funcionalidades da aplicação e em adição a estrutura e ao comportamento usualmente encontrados em objetos convencionais, suportam:

- a declaração de novos tipos de restrições temporais;

- a associação de restrições temporais e manipuladores de exceções à declaração de seus métodos;

- a associação de uma cláusula *timeout* à ativação de métodos.

- o estabelecimento de valores para os atributos das restrições temporais na ativação de métodos;

A estrutura geral de uma classe, a partir da qual os OBTR serão instanciados, é definida como sendo[1]:

OBTR class < *id-classe* >
begin
    < *Seção de declaração de tipos de restrições temporais* >
    < *Seção de declaração de dados* >
    < *Seção de declaração de métodos* >
end

• **Seção de declaração de tipos de restrições temporais** - é o local onde o usuário poderá declarar novos tipos de restrições temporais, os quais em adição aos tipos pré-definidos (*Periodic, Aperiodic e Sporadic*) poderão ser associadas aos métodos do objeto-base que está sendo definido. Cada novo tipo de restrição temporal introduzido a nível de objeto-base, deverá possuir uma implementação no meta-objeto gerenciador correspondente.

A forma geral de declaração de novos tipos de restrições temporais, é a seguinte:

RT-Type <*id-RT*> = [<*tipo-RT*> ,] (<*atributos-RT*>)

---

1 Palavras reservadas serão escritas em inglês e destacadas em negrito.

onde:

- *<id-RT>* identifica a restrição temporal que está sendo introduzida;

- *<tipo-RT>* especifica se a restrição temporal é *time-trigger* ("TT") ou *event-trigger* ("ET"), sendo que "ET" é assumido como *default*. Este atributo determina a forma de ativação dos métodos aos quais a restrição estiver associada: por relógio ("TT") ou por mensagens ("ET");

- *<atributos-RT>* especifica a lista de atributos que compõem a restrição que está sendo declarada; alternativamente esta especificação poderá ser mais elaborada, envolvendo por exemplo uma composição entre tipos de restrições existentes.

A seguir são apresentados alguns exemplos de declarações de tipos de restrições temporais[2]:

RT-Type *ActivationInterval* = (StartTime, EndTime, MET);
// Especifica um intervalo de tempo no qual um método deverá ser executado, onde
// MET (Maximum Execution Time) é o tempo máximo de execução desse método.

RT-Type *TimingPolimorphic* = (Deadline, Met1, Met2, ... , Metn);
// Especifica os diversos métodos que poderão ser ativados em resposta a um
// pedido de ativação do método ao qual esta restrição vier a ser associada.

RT-Type *Start-at-TT* = "TT", (StartTime, Deadline, MET);
// declara uma restrição time-trigger, a qual especifica o instante de tempo
// a partir do qual um método deverá ser ativado.

• **Seção de declaração de dados** - é a seção onde os tipos, variáveis e constantes do objeto deverão ser declaradas como convencionalmente, de acordo com a linguagem utilizada.

• **Seção de declaração de métodos** - é a seção onde os métodos do objeto são declarados e as restrições temporais e manipuladores de exceções são a eles associadas. Os valores dos atributos referentes a uma restrição temporal associada a um método particular, poderão ser estabelecidos no momento da declaração deste método (valores *default*), no momento da criação do objeto ou no momento da ativação do método.

A declaração de um método em um objeto-base será da seguinte forma:

<tipo> < id-método> (<parametros>), <restrição-temporal>, <id-exceção>

onde:

<tipo> - especifica o tipo do valor de retorno;

<id-método> - especifica o nome do método;

<parametros> - declara os parâmetros formais do método;

<restrição-temporal> - especifica o tipo da restrição temporal associado ao método, incluindo seus atributos. Os nomes dos atributos poderão ou não ser os mesmos usados na declaração da restrição em questão e poderão ou não ter valores *default* associados;

<id-exceção> - identifica o manipulador de exceção contendo a ação alternativa a ser executada caso a restrição temporal especificada seja violada.

---

2 Nomes de restrições temporais serão escritos em inglês e destacados em itálico.

Exemplos de declarações de métodos:

1 - **void** Alarme (...), *Aperiodic* (D, MET=10), DesativaSistema( );

2 - **void** VerificaTemperatura (...), *Periodic* (P, Fim, MET=5), RepeteTemp( );

3 - **void** TrataErro (...), *Sporadic* (D, Intervalo-Minimo, MET=10), AcumulaErro( );

4 - **infotype** RecuperaInfo (...), *ActivationInterval* (Inicio, Fim, MET=20), RepeteIinfo();

5 - **imagemtype** RecuperaImagem (...), *TimingPolymorphic* (D, Met1="RIresA", Met2="RIresB"), RepeteImagem( );

imagemtype RIresA (...);

imagemtype RIresB (...);

onde:

1 - Declara "Alarme()" como sendo um método *aperiódico*, cujo *deadline* "D" deverá ser especificado na ativação deste método ou na criação do objeto no qual ele se encontra. Adicionalmente, esta declaração especifica que o tempo máximo de execução (MET - Maximum Execution Time) é de 10ms e que "DesativaSistema()" é o manipulador de exceções associado ao método "alarme()".

2 - Declara "VerificaTemperatura()" como sendo um método *periódico*, onde "P" é o período de ativação e "Fim" especifica o instante limite para a última ativação; os valores de "P" e "Fim" deverão ser estabelecidos no momento da criação do objeto ou no momento da ativação do método. Adicionalmente, é definido um "MET" de 5ms e o manipulador de exceções "RepeteTemp()" é associado ao método declarado.

A semântica associada a esta restrição, dependerá do tipo da restrição temporal pré-definida *Periodic*: se ela for "ET", os métodos aos quais ela estiver associada deverão ser inicialmente ativados por mensagens assíncronas, sendo que as demais ativações serão por relógio; se for "TT", todas as ativações, incluindo a primeira, serão feitas por relógio, e neste caso o instante da primeira ativação deverá ser especificado explicitamente na associação da restrição aos métodos, ou será assumido como sendo o instante da criação do objeto no qual o método se encontra.

3 - Declara "TrataErro()" como sendo um método *esporádico*, cuja diferença básica com relação a um método *aperiódico*, consiste na especificação de um intervalo mínimo ("Intervalo-Minimo") a ser observado entre duas ativações sucessivas do método. Adicionalmente é definido um "MET" de 10ms para execução do método "TrataErro()", ao qual é associado o manipulador de exceções "AcumulaErro()".

4 - Esta declaração especifica um *intervalo de tempo* no qual o método "RecuperaInfo()" deverá ser executado. As informações de tempo de inicio ("StartTime") e tempo de fim ("EndTime") do intervalo deverão ser especificadas no momento de cada ativação. Adicionalmente é definido um "MET" de 20ms e o manipulador de exceções "RepeteInfo()" é associado ao método declarado.

5 - Esta declaração define "RecuperaImagem()" como sendo um método *polimórfico temporal*, especificando que os métodos "RIresA()" e "RIresB()", são as versões que podem ser executadas em resposta a uma ativação de "RecuperaImagem()"; estas versões, supostamente possuem tempo máximo de execução ("MET") distintos e conhecidos. A escolha da versão a ser executada, dependerá da disponibilidade de tempo com relação ao

deadline "D" especificado no momento da ativação. Caso o tempo disponível não seja suficiente para a execução de qualquer das versões especificadas, o manipulador de exceções, identificado por "RepeteImagem()", será ativado.

Ativação de métodos - As ativações de métodos com restrição temporal associada realizadas nos métodos de um OBTR, deverão fornecer os valores reais para os atributos que compõem a restrição temporal em questão. Isto poderá ser realizado, por exemplo, acrescentando-se um segundo conjunto de argumentos (argumentos temporais) aos comandos de ativação convencionais, que passarão a ter a seguinte forma geral:

[<*Id-Objeto*>] . <*Id-Metodo*>(<*Argumentos-Funcionais*>) [,(<*Argumentos-Temporais*>)]

Adicionalmente, para permitir que o objeto chamador (cliente) possa decidir por si próprio o tempo de espera pelo resultado do método solicitado, pode-se associar uma cláusula *timeout* aos comandos de ativação de métodos (com ou sem restrições temporais associadas). A estrutura e a semântica da cláusula timeout são descritas na seção III.7 deste capítulo, onde é apresentada uma proposta de extensão do modelo RTR para ambientes distribuídos abertos.

III.4.1.1 - Exemplo de um OBTR

A figura 3.2 apresenta a definição de uma classe de OBTR, cujos métodos possuem diferentes comportamentos temporais. Nesta definição, as restrições de periodicidade e de aperiodicidade utilizadas são consideradas pré-definidas, enquanto a restrição "*Start-at*" é declarada na seção de declaração de tipos de restrições temporais e deverá ser implementada no meta-objeto gerenciador (MOG) correspondente.

```
OBTR class ClasseExemplo
begin
    // definição de novos tipos de restrições temporais
    RT-Type Start-at = (StartTime, Deadline, MET);
    // declaração das variáveis da classe
    ...
    // declaração dos métodos:
    void Met1 ( ... ), Aperiodic (D, MET=20), ExcMet1( );
    begin ... end;
    void Met2 ( ... ), Periodic (P, Fim, MET=10), ExcMet2 ( );
    begin ... end;
    void Met3 ( ... ), Aperiodic(D, MET=15), ExcMet3 ( );
    begin ... end;
    void Met4 ( ... ), Start-at (Tinicio, D, MET=10), ExcMet4;
    begin ... end;.
end
```

Figura 3.2 - Exemplo de um Objeto-Base Tempo Real

Criação de objetos-base tempo real (OBTR) - A criação de OBTR é feita por instanciação de classes do tipo OBTR; Semanticamente, a criação de um OBTR implicará na criação de um meta-objeto-gerenciador (MOG) correspondente, o qual será uma instância da meta-classe relacionada a classe do OBTR. A ligação entre um OBTR e seu MOG poderá ser

automática, sendo que o nome da meta-classe deverá ser o mesmo da classe OBTR em questão, só que precedido pelo *string* "Meta".

Por exemplo, a partir da classe especificada na figura 3.2, a declaração:

ClasseExemplo ObjetoExemplo;

resultaria na criação de um OBTR denominado "ObjetoExemplo" e de um meta-objeto gerenciador (MOG), denominado "MetaObjetoExemplo" instanciado a partir da classe "MetaClasseExemplo", a qual deverá ser provida pelo usuário. Esta meta-classe deverá conter a implementação da restrição "*Start-at*" e dos manipuladores de exceção ("ExcMet1", "ExcMet2", "ExcMet3", "ExcMet4") associados aos métodos do OBTR "ObjetoExemplo"; além disso, as restrições temporais "*Aperiodic*" e "*Periodic*" deverão ser acessíveis (via herança, importação ou redefinição).

III.4.2 - Meta-Objetos Gerenciadores (MOG)

Estrutura geral - Os meta-objetos gerenciadores são objetos *multi-threads* responsáveis pelo gerenciamento dos pedidos de ativação dos métodos dos objetos-base correspondentes, pelo controle de concorrência em um objeto-base, pelo gerenciamento das restrições de sincronização e pelo processamento das restrições temporais (quando será decidido pela efetiva ativação do método solicitado, ou pelo levantamento da exceção temporal correspondente). Os pedidos de ativação recebidos pelo MOG, executarão em suas próprias *threads* de controle, viabilizando o tratamento concorrente destes pedidos.

A estrutura geral das meta-classes a partir das quais os meta-objetos gerenciadores serão instanciados, é a seguinte:

MOG class *id-meta-classe*
begin
<*Seção de gerenciamento*>
<*Seção de sincronização*>
<*Seção de exceções temporais*>
<*Seção de restrições temporais*>
end

A seguir descrevemos as funções, a estrutura e o comportamento de cada seção, representamos graficamente a interação entre estas seções (figura 3.3) e apresentamos um exemplo completo e comentado de um MOG (figura 3.4).

III.4.2.1 - Seção de gerenciamento

Funcionalidade - A seção de gerenciamento é a seção para a qual serão encaminhados todos os pedidos de ativação dos métodos do objeto-base correspondente e suas funcionalidades básicas são as seguintes:

- *Receber pedidos de ativação* provenientes do redirecionamento das ativações de métodos endereçadas ao objeto-base correspondente (ação 1 das figuras 3.3a e 3.3b), verificando para cada método solicitado se ele possui ou não restrições temporais associadas. Caso possua, será ativado o método da seção de restrições temporais que implementa a restrição temporal em questão (ação 2 da figura 3.3a), o qual será responsável pelo processamento do pedido a partir deste ponto. Caso o método solicitado não possua restrição

temporal associada, será ativado o método "ProcessaPedidoSemRestricaoTemporal()", da própria seção de gerenciamento, que ficará responsável pelo processamento do pedido em questão. O comportamento do MOG relativo a este caso é mostrado na figura 3.3b.

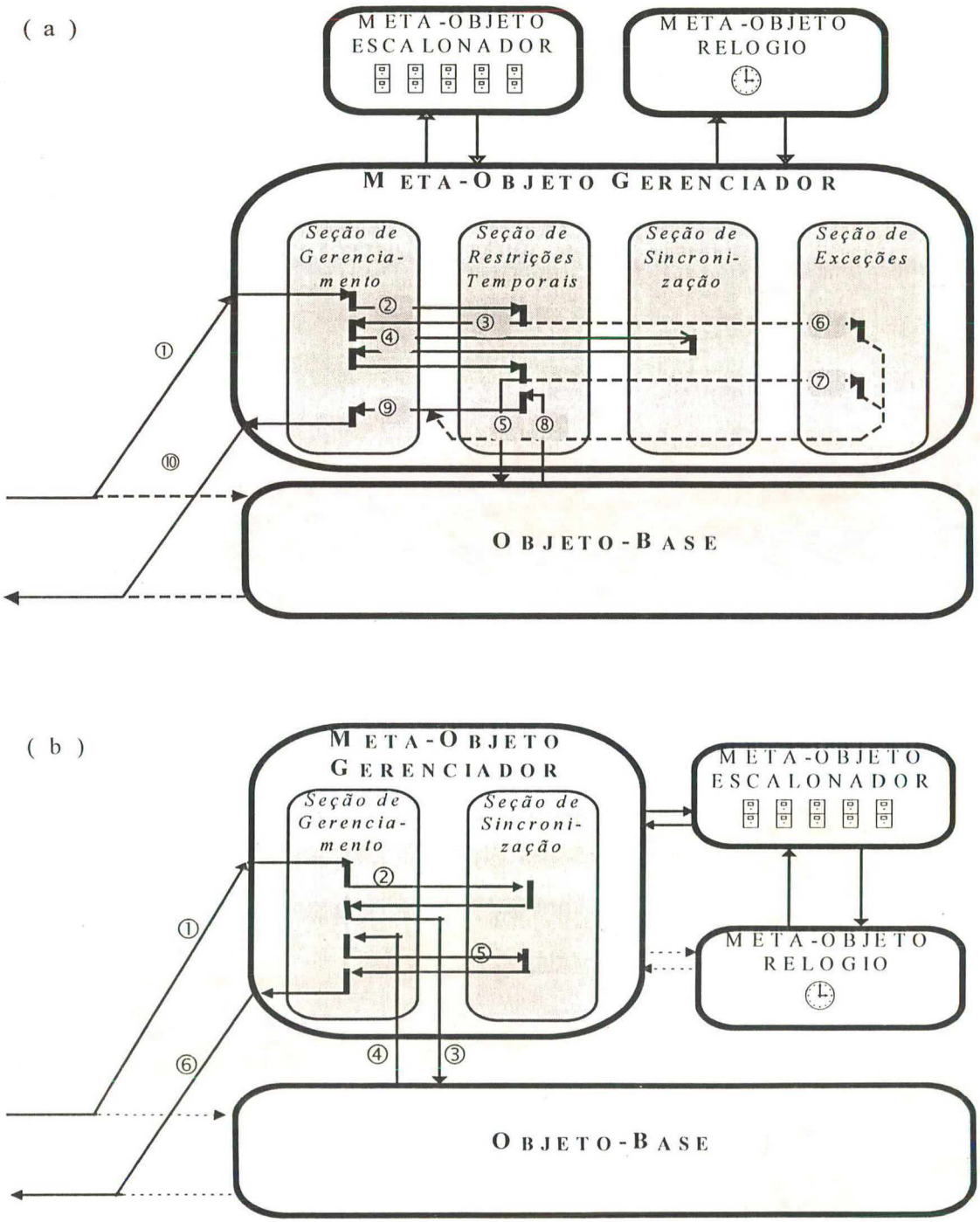


Figura 3.3 - Dinâmica interna do Meta-Objeto Gerenciador relativa ao tratamento de (a) métodos com restrição temporal e (b) métodos sem restrição temporal.

- *Controlar a concorrência* no objeto-base correspondente, garantindo que seus métodos executem de forma mutuamente exclusiva, de acordo com a ordem de execução estabelecida pelo meta-objeto escalonador.

Estrutura e comportamento - A seção de gerenciamento será composta por métodos que implementam as funcionalidades acima descritas, por uma variável que mantenha o estado de execução do objeto-base ("ativo" ou "dormindo") e por uma fila, denominada "FilaDePendencias", na qual serão colocados todos os pedidos de ativação que embora tendo sido liberados pelo meta-objeto escalonador, não puderam ser ativados porque o objeto-base encontrava-se em estado "ativo" ou porque não satisfizeram, no momento da liberação, as restrições de sincronização especificadas. A seguir, descrevemos o comportamento básico dos métodos que devem compor a seção de gerenciamento de um MOG:

• *RecebePedido()* - Ao ser ativado este método verifica se o método solicitado possui ou não restrição temporal associada; esta verificação pode ser realizada reflexivamente (se a linguagem utilizada suportar reflexão estrutural) ou convencionalmente através de um descritor do objeto-base mantido explicitamente pelo MOG. Dependendo desta verificação, será ativado o método que implementa a restrição temporal existente, ou o método que trata os pedidos sem restrições temporais.

• *ProcessaPedidoSemRestriçãoTemporal()* - O comportamento deste método consiste inicialmente em encaminhar um pedido de escalonamento do método solicitado ao meta-objeto-escalonador (MOE) e aguardar o retorno deste pedido com autorização para ativação do método em questão. Ao receber esta autorização deverão ser verificadas as condições de concorrência e de sincronização (ação 2 da figura 3.3b) e, quando estas forem satisfeitas, deverá ser efetivada a ativação do método solicitado no objeto-base (ação 3). Após o término da execução do método solicitado (ação 4), deverá ser atualizado o estado de sincronização do objeto-base (ação 5) e em seguida o controle da execução deverá ser retornado para o objeto-base que efetuou a solicitação (ação 6).

•*LiberaPedidoDeAtivação() e FimDeExecução()* - Conjuntamente estes métodos implementam o controle de concorrência do modelo, sendo que "LiberaPedidoDeAtivação()" será ativado a partir de outros métodos do MOG após os pedidos que estão sendo analisados tiverem sido liberados para ativação pelo meta objeto escalonador, e "FimDeExecução()" será ativado sempre que for concluída a execução de um método liberado anteriormente. Este controle (ações 3 e 9 da figura 3.3a) será feito com base no estado do objeto-base mantido pelo MOG, o qual poderá ser "ativo" ou "dormindo". Considerando-se o caso de métodos com restrição temporal, este controle pode ser assim explicitado:

- quando o estado do objeto-base for "ativo" (i.e., um de seus métodos esta sendo executado), os pedidos que chegarem serão colocados em uma fila de pendências ("FilaDePendencias"), de onde serão retirados quando a execução do método corrente for concluída;

- quando o estado do objeto-base for "dormindo", se o método solicitado puder ser executado no atual estado de sincronização do objeto-base (o que deverá ser verificado através de uma interação com a seção de sincronização - ação 4 da fig. 3.3a), sua ativação deverá ser efetuada (ação 5 da fig. 3.3a) e o estado do objeto-base deverá ser alterado para "ativo";

- quando concluído o processamento de um pedido (ação 8 da fig. 3.3a), será liberado o próximo pedido da fila de pendências, de acordo com uma política dependente da implementação do modelo. Caso a fila esteja vazia, o estado do objeto muda para "dormindo", permanecendo neste estado até a chegada de um novo pedido, e o MOE deverá ser notificado para que um novo pedido possa ser escalonado.

III.4.2.2 - Seção de sincronização

Esta seção é opcional e nela são definidas e controladas as restrições de sincronização entre os métodos do objeto-base, com o propósito de influenciar o processamento de pedidos de acordo com a lógica da aplicação. Mecanismos como "*path-expression*" [Campbell 74], "*set-enables*" [Tomlinson 89] e "máquinas de estado", entre outros, podem ser usados para este propósito. A estrutura e o comportamento desta seção, dependerá do mecanismo utilizado para expressar as restrições de sincronização.

Por exemplo, se usado o mecanismo "*Path-Expression*", a seção de sincronização será composta pela descrição das restrições de sincronização na forma de uma "*path-expression*" (onde os operandos seriam os métodos do objeto-base) e por métodos auxiliares usados para verificar se um determinado método está ou não apto a ser executado no atual estado de sincronização do objeto-base correspondente, e para atualizar o estado de sincronização do objeto-base. Estes métodos devem ser ativados, respectivamente, antes da ativação do método do objeto-base e imediatamente após a sua execução.

O mecanismo de "*path-expression*" é particularmente interessante por sua efetividade, elegância e sobretudo pela sua adequação a filosofia reflexiva, onde as restrições de sincronização são naturalmente separadas do código da aplicação. Entretanto, o uso de outros mecanismos de sincronização existentes ou definidos pelo usuário também podem vir a ser utilizados.

III.4.2.3 - Seção de exceções temporais

Nesta seção são implementados os manipuladores das exceções relativas a violação das restrições temporais especificadas nos métodos do objeto-base; para cada manipulador de exceção temporal especificado na declaração dos métodos do objeto-base, deverá haver uma implementação (na forma de um método) nesta seção. A ativação dos manipuladores de exceções temporais se dará a partir dos métodos que implementam as restrições temporais, sempre que for prevista ou constatada a violação de uma restrição temporal (ações 6 e 7 da figura 3.3a).

III.4.2.4 - Seção de restrições temporais

Nesta seção são implementados os novos tipos de restrições temporais (um método para cada restrição) declarados no objeto-base correspondente. A redefinição de tipos de restrições temporais pré-definidos, também é possível, e neste caso a nova implementação também deverá ser introduzida nesta seção. A ativação de um método implementando uma restrição temporal se dará a partir do seção de gerenciamento (ação 2 da figura 3.3a), sempre que esta receber um pedido de ativação de um método com restrição temporal associada.

Para processar as restrições temporais, o meta-objeto gerenciador (MOG) interage (através dos métodos que implementam as restrições temporais) com o meta-objeto escalonador, o qual determinará, de acordo com a política de escalonamento implementada, o momento a partir do qual o pedido de ativação que está sendo analisado poderá ser liberado para execução. Neste momento, se houver disponibilidade de tempo para execução do método liberado, serão verificadas as questões de concorrência e sincronização (através da interação com as seções de gerenciamento e de sincronização - ações 3 e 4 da figura 3.3a), caso contrário será levantada uma exceção temporal (ação 6 da figura 3.3a).

Após verificadas as questões de concorrência e sincronização, se a restrição temporal ainda não tiver sido violada, o método do objeto-base solicitado será ativado (ação 5 da figura 3.3a) com parâmetros de escalonamento que priorizem o atendimento desta solicitação; caso contrário uma exceção temporal será levantada (ação 7 da figura 3.3a). Concluída a execução do método solicitado (ação 8 da figura 3.3a) ou da restrição temporal executada em seu lugar, o controle da execução volta para a seção de gerenciamento (ação 9) de onde retorna para o objeto que efetuou a requisição (ação 10 da figura 3.3). Adicionalmente, dependendo da restrição temporal em questão, o MOG também poderá interagir com o meta-objeto relógio (MOR), para efetivar o processamento destas restrições temporais (por exemplo, programando ativações futuras).

III.4.2.5 - Exemplo de um Meta-Objeto Gerenciador

Visando explicitar a composição de um MOG e melhor explicar o inter-relacionamento entre o MOG e os demais elementos do modelo (figura 3.1) e entre as seções internas do MOG (figura 3.3a), apresentamos na figura 3.4 um exemplo de como poderia ser especificado o MOG correspondente ao OBTR apresentado na figura 3.2 e comentamos o seu comportamento a partir de uma suposta ativação do método "Met1()" do objeto-base considerado.

O comportamento do MOG especificado na figura 3.4, considerando-se uma ativação do método "Met1" do objeto-base "ObjetoExemplo", será o seguinte:

1 - Quando ocorrer um pedido de ativação do método "Met1()" do OBTR "ObjetoExemplo", este pedido será transformado (pelo compilador ou pelo suporte subjacente) em uma chamada ao método "RecebePedido()" do "MetaObjetoExemplo".

2 - O método "RecebePedido()" constatando a existência da restrição temporal "aperiodic" associada ao método "Met1()", ativará o método "*Aperiodic()*" da seção de restrições temporais.

3 - O método "*Aperiodic()*" terá o comportamento especificado na figura 3.5, interagindo com o MOE, com outros métodos do MOG ("LiberaPedidoDeAtivacao()" e "FimDeExecucao()", por exemplo) e com o OBTR "ObjetoExemplo" para efetivar a ativação do método "Met1()".

4 - O método "LiberaPedidoDeAtivacao()" da seção de gerenciamento do MOG é ativado a partir do método "*Aperiodic()*", imediatamente após este ter recebido (do MOE) autorização para ativação do método solicitado ("Met1()").

5 - O método "VerificaSincronização()" da seção de sincronização do MOG é ativado a partir do método "LiberaPedidoDeAtivacao()", antes da decisão de liberar ou não a ativação de "Met1()".

6 - Se as condições de concorrência e sincronização forem satisfeitas, a ativação de "Met1()" é realizada imediatamente; caso contrário o pedido de ativação é colocado na fila de pendências do MOG e o MOE será notificado para que um novo pedido possa ser liberado.

7 - O método "AtualizaEstadoDeSincronizacao()" da seção de sincronização do MOG é ativado após o término da execução do método "Met1()" e sua função é atualizar o estado de sincronização do objeto-base, considerando a execução deste método.

8 - O método "FimDeExecucao()" da seção de gerenciamento do MOG é ativado a partir do método "*Aperiodic()*" após o término da execução de "Met1()" ou da exceção executada em seu lugar.

```
MOG class MetaClasseExemplo;
begin
   // Declaração das variáveis da classe, incluindo descritor do objeto-base,
   // estado de execução e estado de sincronização do objeto-base e a fila
   // de pedidos pendentes
   // *** seção de gerenciamento ***
   void RecebePedido (MetId, ... )
   begin ... end;
   void ProcessaPedidoSRT (MetId, ... )
   begin ... end;
   void LiberaPedidoDeAtivacao (MetId)
   begin ... end;
   void FimDeExecucao(MetId)
   begin ... end;
   // *** seção de sincronização ***
   Path 2 : (Met1 ; Met3) end;
   bool VerificaSincronizacao(MetId)
   begin ... end;
   void AtualizaEstSincronizacao (MetId)
   begin ... end;
   // *** seção de exceções temporais ***
   void ExcMet1 ( ... ) begin ... end;
   void ExcMet2 ( ... ) begin ... end;
   void ExcMet3 ( ... ) begin ... end;
   void ExcMet4 ( ... ) begin ... end;
   // *** seção de restrições temporais ***
   void Periodic(MetId, ExcId, P, Fim, MET)
   begin ... end;
   void Aperiodic (MetId, ExcId, D, MET)
   begin ... end;
   void Start-at (MetId, ExcId, Tinicio, D, MET)
   begin ... end;
end;
```

Figura 3.4 - Exemplo de um Meta-Objeto Gerenciador

Em geral, os métodos que compõem o MOG poderão ser herdados (ou importados) de classes pré-definidas, com exceção dos aspectos dependentes da aplicação, tais como os manipuladores de exceção e as restrições temporais introduzidas no objeto-base correspondente. Alternativamente, o usuário poderá redefinir total ou parcialmente os procedimentos referentes às seções de gerenciamento, de sincronização e de restrições temporais, visando a obtenção de um comportamento mais adequado às especificidades da aplicação em questão (por exemplo, modificando o tratamento dado aos pedidos de ativação de métodos sem restrição temporal e substituindo a política usada no gerenciamento da fila de pendências).

```
void Aperiodic (MetId, ExcId, D, MET)
// MetId - Identificador do método solicitado
// ExcId - Identificador do manipulador de exceções associado a MetId
// D - "deadline" estabelecido para a ativação corrente de MetId
// MET - Tempo máximo de execução de MetId
begin
    // Solicita escalonamento de MetId ao MOE
    id-MOE.Escalona (id-MOG, MetId, D)
    // Após autorização do MOE para ativação, verifica se D pode ser satisfeito
    If D > (current-time + MET)
    Then
        // Verifica concorrência e sincronização
        id-MOG.LiberaPedidoDeAtivação (MetId);
        // Ao retornar, verifica se D ainda pode ser satisfeito
        If D > (current-time + MET)
        Then
            // Ativa método solicitado do OBTR correspondente
            id-OBTR.MetId ( ... );
            // Atualiza estado de sincronização
            id-MOG.AtualizaEstadoSincronizacao(MetId)
        Else
            // deadline violado enquanto MetId aguardava na FilaDePendencias
            id-MOG.ExcId ( ... )
        End if
        // Informa seção de gerenc. que um novo pedido pode ser liberado
        id-MOG.FimDeExecução ( ... )
    Else
        // Quando MetId foi escalonado, já não havia tempo para sua execução
        id-MOG.ExcId ( ... );
        // Libera MOE para o escalonamento do próximo pedido
        id-MOE.LiberaProximoPedido ( )
    End if
end;
```

Figura 3.5 - Exemplo de implementação da restrição temporal "aperiodic"

III.4.3 - Meta-Objeto Escalonador (MOE)

Funcionalidade - O meta-objeto escalonador (MOE) tem como função básica receber, ordenar e liberar pedidos de escalonamento dos diversos meta-objetos-gerenciadores que compõem a aplicação, de maneira "on-line" e de acordo com uma determinada política de escalonamento. Através do MOE, os diferentes pedidos de escalonamento originados dos diversos meta-objetos-gerenciadores que compõem uma aplicação, podem ser analisados globalmente e ordenados de acordo com uma política de escalonamento adequada às especificidades da aplicação.

A existência de um MOE acessível ao projetista/programador, permite que diferentes políticas de escalonamento sejam implementadas independentemente do suporte. Para tanto, o

MOE deverá mapear suas decisões de escalonamento para o suporte de execução existente (em função do esquema de prioridades existente), de forma a garantir que estas decisões sejam consideradas na execução da aplicação.

Além das funções básicas acima estabelecidas, que configuram um MOE básico, uma implementação particular do MOE, poderá também suportar outras funções relativas ao escalonamento tempo-real de tarefas, tais como:

- Combinar diferentes políticas no escalonamento de diferentes classes de tarefas tempo real;

- Realizar análise de escalonabilidade dinâmica das tarefas do sistema como um todo ou de determinada classe de tarefas, antecipando a detecção de violações temporais daquelas tarefas que não poderão ser escalonadas e garantindo a satisfação das restrições temporais das tarefas aceitas;

- Substituir dinamicamente a política de escalonamento usada, em função da evolução da execução do sistemas e de acordo com informações obtidas em tempo de execução.

No caso de aplicações centralizadas haverá um único MOE. Nas aplicações distribuídas existirá um MOE em cada nodo onde a aplicação estiver sendo executada; neste caso, os diversos MOE's atuarão independentemente, influenciando apenas o escalonamento local.

Estrutura - A estrutura básica de um MOE é apresentada na figura 3.6. Um MOE básico é composto pelos métodos "Escalona()", "LiberaProximoPedido()" e "RetiraDaFila()", os quais operam sobre uma fila de escalonamento e controlam o estado da execução da aplicação.

```
MOE class id-MOE
begin
    // variáveis básicas
    FilaDeEscalonamento = "vazia"
    Estado = "dormindo"
    // métodos básicos
    void Escalona (id-MOG, MetId, ... )
    begin ... end;
    void LiberaProximoPedido ( ... )
    begin ... end;
    void RetiraDaFila (MetId)
    begin ... end
end; // class id-MOE
```

Figura 3.6 - Estrutura geral de um Meta-Objeto Escalonador

Estes métodos podem ser assim descritos:

- O método "Escalona()" implementa a política de escalonamento a ser utilizada, sendo responsável pelo recebimento de pedidos de escalonamento e pela ordenação destes pedidos de acordo com a política implementada; adicionalmente, para cada pedido enfileirado, o meta-objeto relógio (MOR) poderá ser programado para controlar o seu *deadline*;

- O método "LiberaProximoPedido()", é responsável pelo controle do estado de execução da aplicação e pela liberação do próximo pedido a ser executado, com parâmetros de escalonamento que garantam seu escalonamento imediato por parte do suporte subjacente. Adicionalmente, no caso de uso de políticas preemptivas e de aplicações distribuídas, será função deste método controlar a prioridade dos pedidos liberados (para que não ocorra inversão de prioridade) e evitar que o nodo fique ocioso durante a execução de tarefas remotas;

- O método "RetiraDaFila()", será ativado pelo meta-objeto relógio (MOR) sempre que for constatada a violação do *deadline* (ou de outro atributo temporal utilizado) de um pedido de escalonamento que encontra-se na fila de escalonamento; sua função básica é, dependendo da política em uso, tentar antecipar a execução da ação alternativa (exceção temporal) correspondente a violação temporal detectada ou simplesmente registrar informações que subsidiem uma avaliação da política em uso e dos parâmetros temporais utilizados.

III.4.4 - Meta-Objeto Relógio (MOR)

Funcionalidade - O MOR é uma abstração do relógio do sistema, estruturada na forma de objeto; sua função básica é receber pedidos para ativar métodos num tempo futuro e efetuar estas ativações no tempo programado. Adicionalmente, o MOR é utilizado pelo MOE para detectar eventuais violações de *deadline* dos métodos cujos pedidos de escalonamento estão aguardando para serem escalonados. A exemplo do MOE, existirá um MOR em cada nodo onde a aplicação estiver sendo executada.

Estrutura - A estrutura geral da meta-classe a partir da qual um meta-objeto relógio deve ser instanciado, é mostrada na figura 3.7; segundo esta estrutura, um MOR é composto por três métodos que atuam sobre uma fila de pedidos ("FilaPedidosFuturos") na qual são armazenados todas as solicitações de futuras ativações.

```
MOC class id-Clock      // Meta Objeto Clock
begin
    // variável básica
    FilaPedidosFuturos = vazia;  // ordenada por tempo de ativação
    // métodos
    void ProgramaAtivacao (T, id-MO, MetId(...))
    begin ... end;
    void EfetuaAtivacao ( )
    begin ... end;
    void CancelaProgramacao (id-MO, MetId)
    begin ... end;
end  // class id-MOC
```

Figura 3.7 - Estrutura geral de um Meta-Objeto Relógio

Os métodos e suas funcionalidades são os seguintes:

- "ProgramaAtivacao()", ativado a partir de um MOG (quando do processamento de uma restrição temporal) ou do MOE (quando do enfileiramento de um pedido de escalonamento) com o objetivo de "programar" uma ativação futura;

- "EfetuaAtivacao", ativado automaticamente pelo suporte (em intervalos regulares), a função básica deste método é verificar a existência de ativações programadas para o tempo corrente e, se for o caso, realizar estas ativações; na prática este método controla a passagem do tempo, notificando que uma determinada quantidade de tempo esgotou-se (podendo significar, por exemplo a violação de um *deadline*) e realizando ativações *time-trigger* nos tempos previamente programados;

- "CancelaProgramacao", ativado a partir do MOG ou do MOE, a função deste método é cancelar a programação de ativações para um tempo futuro.

III.5 - Explorando o potencial e demonstrando a expressividade do modelo RTR

III.5.1 - Introdução

Além de suas facilidades inerentes, o modelo RTR suporta adicionalmente a representação e o controle de várias situações tipicamente encontradas no desenvolvimento de STR, sem que sua estrutura e sua semântica de funcionamento tenham que ser modificadas. O suporte a tais funcionalidades, seguindo a filosofia reflexiva do modelo, é realizado no meta-nível da aplicação, através da redefinição (extensão, alteração ou substituição) das funções básicas dos MOG e do MOE, que como previsto originalmente são acessíveis ao programador da aplicação.

Nesta seção identificamos várias situações que podem ser representadas e controladas adequadamente pelo modelo RTR, demonstrando assim sua potencialidade, sua expressividade e seu alto grau de adaptabilidade; as situações consideradas estão relacionadas a: expressividade (III.5.2 e III.5.3), ajuste dinâmico dos atributos das restrições temporais (III.5.4 e III.5.5), escalonamento tempo real (III.5.6 a III.5.8) e gerenciamento de memória (III.5.9). Adicionalmente, a expressividade do modelo RTR na representação de restrições de sincronização multimídia, será apresentada e exemplificada na próxima seção (III.6).

III.5.2 - Refletindo aspectos não temporais

Além dos aspectos temporais, de concorrência e de sincronização que são inerentemente reflexivos na filosofia RTR, outros aspectos comportamentais da aplicação também podem ser refletidos sem que o esquema básico de reflexão (estrutura e semântica de funcionamento) precisem ser modificados. Como exemplo de tais comportamentos podemos citar: depuração, controle estatístico, persistência, qualidade de serviço etc. Para isso é necessário explicitar que tipos de controle devem ser realizados sobre quais métodos; ou seja, é necessário agrupar os métodos por categorias de controle, sendo que a associação de uma categoria a um método, determina os procedimentos de controle a serem realizados quando este método for ativado.

Para obtermos este comportamento no modelo RTR (onde a operação de ativação de métodos já é reflexiva), podemos adicionar uma cláusula **"categoria"** a declaração dos métodos do objeto-base como mostrado no exemplo abaixo.

<*tipo*> Id-Metodo (...), <*restrição-temporal*>,
<*exceção-temporal*>,
categoria = < *identificador-da-categoria*>

Adicionalmente, o comportamento correspondente a categoria em questão deverá ser executado sempre que tais métodos forem requisitados; tal comportamento poderá ser implementado através de métodos auxiliares ou embutido no método "RecebePedido()" do meta-objeto gerenciador. Dependendo das particularidades de cada categoria, este controle poderá ser executado antes ou depois da execução do método requisitado.

Controles como estes, podem ser usados com diferentes finalidades. Por exemplo, controle de depuração pode ser útil durante a fase de testes do sistema, enquanto que controles estatísticos e de QoS podem ser usados para influenciar dinamicamente (ou mesmo modificar) o comportamento do sistema. Por exemplo, o número de violações temporais ocorridas poderia ser usado como base para decisões relativas ao ajuste dos atributos temporais da tarefa ou mesmo a mudança da política de escalonamento.

Aprofundando esta idéia, o esquema proposto (ou um esquema similar estendido) poderia ser usado para permitir reflexão de comportamentos mais complexos tais como tolerância a faltas e sincronização multimídia por exemplo, possivelmente através de meta-objetos adicionais subordinados ao meta-objeto gerenciador. Esquemas similares são usados em outros modelos reflexivos propostos, tais como R^2 [Honda 94], onde o próprio controle temporal é realizado por um meta-objeto separado e FRIENDS [Fabre 96], que trata diferentes comportamentos (tolerância a faltas, distribuição e segurança) usando meta-objetos especializados para cada comportamento refletido.

III.5.3 - Polimorfismo temporal

Segundo [Lin 91], em muitas situações práticas é preferível ter-se uma solução aproximada dentro do tempo especificado do que uma solução exata porém fora do tempo especificado. A técnica usada na programação destas situações denomina-se computação imprecisa, a qual caracteriza-se pela realização do que é possível dentro do tempo disponível. Esta técnica tem sido utilizada por linguagens de programação tempo real tais como Flex [Lin 91] e DROL [Takashio 92].

Da mesma forma que Flex e DROL, o modelo RTR também permite a realização de computação imprecisa baseada na técnica de programação denominada N-versões; segundo esta técnica, uma determinada tarefa da aplicação pode possuir diferentes versões com diferentes tempos de execução (onde a qualidade ou a precisão dos resultados obtidos é proporcional ao tempo de execução das versões disponíveis). Em resposta a ativação de uma tarefa, uma de suas versões será escolhida dinamicamente para execução em função do tempo disponível no momento do escalonamento da tarefa solicitada. O termo "polimorfismo temporal" é atribuído a esta situação em função de sua similaridade com o conceito de polimorfismo tipicamente encontrado no paradigma de objetos.

Em uma aplicação modelada segundo a filosofia do modelo RTR, esta funcionalidade pode ser assim obtida e exemplificada:

- definição de um novo tipo de restrição temporal;

RT-Type *TimingPolymorphic* = (Deadline, <MethodList>);

- associação desta restrição temporal a um método do objeto-base;

void DisplayImagem(...), *TimingPolymorphic* (D, Met1="DI-qA",
Met2=" DI-qB", Met3=" DI-qC"),
IdExcecaoTemporal ( )

- disponibilidade no objeto-base de n-versões com diferentes tempos de execução implementando a mesma funcionalidade;

```
void DI-qA ( ... ) // MET = 150
begin ... end;
void DI-qB ( ... ) // MET = 100
begin ... end;
void DI-qC ( ... ) // MET = 50
begin ... end;
```

- implementação, a nível de meta-objeto gerenciador, da restrição temporal introduzida, cuja funcionalidade básica é escolher dinamicamente uma das versões existentes, sempre que o método "DisplayImagem()" for ativado. Deve ser notado que no exemplo apresentado as diferentes versões do método "DisplayImagem()" foram especificadas estaticamente na declaração deste método, enquanto que o atributo *deadline* ("D") permaneceu variável, devendo ser especificado explicitamente a cada ativação do método "DisplayImagem()". Na prática, entretanto, as versões disponíveis também poderiam ser especificadas dinamicamente em cada ativação do método em questão.

A noção de polimorfismo temporal, embora seja mais geral, também pode ser usada para implementar o esquema de escalonamento "*Task-Pair*" [Mitchell 97]. Este esquema consiste em considerar uma tarefa tempo real com *deadline* "D" como sendo um par de tarefas, onde o primeiro componente, denominado *hard*, possui um tempo de execução de pior caso (*worstcase)* conhecido e o outro, denominado *soft*, possui um tempo de execução desconhecido ou estimado de forma pessimista. Para tanto, uma tarefa teria duas versões (*hard* e *soft*), sendo que a versão a ser executada seria escolhida dinamicamente em função da disponibilidade de tempo no momento da ativação.

Alternativamente, esta situação pode ser implementada como proposto em [Mitchell 97], onde inicialmente a tarefa *soft* é liberada e executará até terminar ou até que o tempo disponível seja suficiente apenas para execução da tarefa *hard.* A flexibilidade do modelo RTR, derivada do uso de reflexão, permite que diferentes implementações da mesma restrição sejam realizadas, sem afetar o código base da aplicação e sem depender do suporte subjacente.

III.5.4 - Ajuste dinâmico do tempo de execução das tarefas

O conhecimento prévio do tempo máximo de execução (MET - Maximum Execution Time) das tarefas de um STR é de fundamental importância para a previsibilidade do sistema; em função disto, STR *hard* necessariamente (ou pelo menos preferencialmente) devem trabalhar com tempo de execução de pior caso (*worstcase)*, enquanto STR *soft* toleram o uso de tempos médios. Atualmente existem muitos problemas na obtenção do *worstcase* (via cálculo) e mesmo dos tempos médios (via medição); problemas estes que tendem a agravar-se na medida em que STR requerem mais flexibilidade, integração com sistemas não tempo real e independência de ambiente operacional.

Neste contexto, o modelo RTR pode contribuir na solução destes problemas permitindo que o tempo de execução das tarefas (métodos) possa ser ajustado dinamicamente em função de medições realizadas de forma reflexiva durante a operação do sistema, de forma transparente ao programador dos objetos-base da aplicação.

Para tanto, a partir de uma estimativa inicial (decorrente da experiência do programador, de medições isoladas e/ou de cálculos parciais) o tempo máximo de execução

das tarefas pode ser redefinido com base no tempo realmente gasto na execução das mesmas; os procedimentos de medição e ajuste podem ser realizados como funções dos métodos que implementam as restrições temporais associadas aos métodos cujo tempo de execução está sendo ajustado. Dependendo da aplicação, todos os métodos com restrições temporais podem ser alvo deste ajuste ou apenas métodos com determinada restrição temporal associada (por exemplo, periodicidade); alternativamente, pode ficar a cargo do programador da aplicação definir os métodos que necessitam ter seu tempo de execução ajustado dinamicamente.

A fórmula a ser usada para o ajuste pode ser estabelecida genericamente com base em informações estatísticas convencionais (tais como média e desvio padrão) relativas ao tempo das últimas execuções do método, ou ainda pode ser especializada para determinados métodos, objetos ou ambientes operacionais. Em qualquer caso, a experiência do projetista e a existência de heurísticas relacionadas também poderão ser consideradas.

III.5.5 - Ajuste dinâmico dos atributos das restrições temporais

Embora normalmente os valores dos atributos das restrições temporais de uma aplicação tempo real sejam inerentes a própria aplicação, nem sempre esses valores são absolutos; em muitos casos uma certa variação pode ser suportada sem comprometer o resultado da aplicação. Além disso, existem casos em que os valores atribuídos inicialmente são meras estimativas que, por força do uso, acabam tornando-se padrão. Contudo, em função das limitações das construções utilizadas para representar as restrições temporais em muitos modelos e linguagens tempo real existentes, tais valores acabam sendo fixados como se fossem absolutos, não suportando nenhum ajuste dinâmico, independentemente da aplicação tolerar ou não determinadas variações.

Um exemplo típico desta situação na área de multimídia, é a taxa de apresentação de mídias contínuas (tais como número de quadros de imagem por segundo ou número de pacotes de som por segundo). Enquanto muitos autores fixam um período de 40 ms (25 quadros/segundo) para apresentação de um quadro de imagem [Blakowski 96], outros estimam que este período pode variar entre 35 e 45 ms (entre 22 e 28 quadros/segundo) sem que a qualidade da apresentação seja prejudicada [Horn 92]. Além disto, os próprios padrões de sistemas de televisão existentes estabelecem diferentes taxas de apresentação; por exemplo o padrão NTSC trabalha com uma taxa de 30 quadros/segundo, enquanto o padrão PALM trabalha com uma taxa de 25 quadros/segundo [Little 94].

Usando mecanismos convencionais para representação de restrições temporais, um valor deve ser escolhido arbitrariamente, e se este valor não se mostrar satisfatório, só poderá ser ajustado em uma próxima execução da aplicação e assim sucessivamente até que um valor adequado seja obtido; este processo pode ser demorado e até mesmo não convergir em função de possíveis variações no sistema e mesmo no ambiente operacional.

No modelo RTR, tanto as variações previsíveis quanto as não previsíveis podem ser facilmente representadas. No primeiro caso, podemos definir uma restrição temporal que represente explicitamente essa variação, fazendo com que o ajuste desejado (embutido no código do método que implementa a restrição) seja realizado dinamicamente a cada ativação dos método ao qual a restrição estiver associada. Para exemplificar esta situação, apresentamos a seguir a definição e o uso de uma restrição de periodicidade na qual o período especificado suporta uma determinada faixa de variação.

```
...
RT-Type PeriodicWithVariation = (Period, Vinf, Vsup, EndTime, MET);
...
void DisplayImagem ( ... ), PeriodicWithVariation (P, V1, V2, F, MET=10),
                                RepeteImagem ( );
begin ... end;
...
Objeto.DisplayImagem ( ... ), (40, 5, 5, 40000);
...
```

Esta restrição temporal, permite um ajuste do atributo *período,* o qual pode variar entre (Período - Vinf) e (Período + Vsup); este ajuste pode, por exemplo, ser usado para compensação de *jitter*. Desta forma, o período de execução do método "DisplayImagem()", variará de 35 a 45ms, dependendo do tempo gasto na execução do mesmo. O valor do período será ajustado para 35ms se a execução for concluída em menos de 35ms, ou para um valor entre 35 e 45ms, dependendo do tempo no qual a execução do método "DisplayImagem()" for concluída. Por outro lado, se o *display* não puder ser efetivado dentro do período máximo (45ms), uma exceção temporal deverá ser levantada, isto é, o manipulador de exceções "RepeteImagem()" deverá ser ativado.

No segundo caso, quando uma variação é aceitável porém não previsível, o ajuste poderá ser feito de maneira similar ao ajuste do tempo máximo de execução, com base em medições efetuadas dinamicamente e com o ajuste sendo realizado através de alguma fórmula baseada nos resultados das medições efetuadas.

III.5.6 - Análise de escalonabilidade dinâmica

Análise de escalonabilidade é um procedimento vital para análise do comportamento temporal de sistemas tempo real. Este procedimento permite que se saiba previamente se um determinado STR (ou parte dele) é ou não escalonável, isto é, se ele pode ou não ser executado de forma que todas as restrições temporais das tarefas que o compõem sejam satisfeitas. Esta análise pode ser estática (realizada *off-line*) ou dinâmica (on-line) e depende de uma série de fatores, tais como técnica de escalonamento utilizada e conhecimento prévio das características temporais das tarefas (tempo de execução, restrições temporais, restrições de precedência, ...) e do ambiente operacional no qual o sistema está sendo executado.

STR *hard*, normalmente requerem a realização de análise de escalonabilidade como um requisito para liberação do sistema, sendo que esta análise deve ser realizada estaticamente *(off-line)*. Por outro lado, um número considerável de STR atuais caracterizam-se por serem flexíveis e dinâmicos, e geralmente não suportam análise estática; entretanto, ainda assim é importante que estes sistemas (ou partes dele) possam ter sua escalonabilidade analisada dinamicamente, objetivando antecipar a detecção de violações temporais e permitir que ações alternativas possam ser executadas mais cedo. A abordagem de escalonamento que suporta este tipo de análise denomina-se "escalonamento baseado em planejamento" [Stankovic 94].

Esta abordagem pode ser representada no modelo RTR através da redefinição do meta-objeto escalonador, adicionando-se a política de escalonamento existente, uma política de admissão de tarefas, segundo a qual só seriam aceitos pedidos de escalonamento com chances de serem executados dentro de suas especificações temporais e sem comprometer a escalonabilidade das tarefas admitidas anteriormente. Neste caso, o uso de reflexão é particularmente interessante por facilitar e flexibilizar a definição de diferentes políticas de

admissão, independentemente do suporte subjacente. Adicionalmente, a filosofia reflexiva do modelo RTR permite que diferentes políticas de admissão sejam combinadas com diferentes políticas de escalonamento, de maneira a melhor satisfazer os requisitos temporais da aplicação.

Para tanto, além de conhecer os atributos temporais das tarefas (disponíveis via esquema reflexivo utilizado), o meta-objeto escalonador (MOE) terá que ser capaz de gerenciar a diferença existente entre o tempo previsto e o tempo efetivamente gasto na execução das tarefas admitidas. Além disso, nos casos onde diferentes aplicações podem ser executadas simultaneamente, o MOE deverá conhecer a quantidade de tempo de processamento disponível para cada aplicação.

Um problema intrínseco, por outro lado, é a dificuldade em se computar precisamente o tempo gasto nas computações realizadas no nível-meta, que além do tempo gasto com o escalonamento inclui também o tempo gasto nas seções de gerenciamento e de restrições temporais do meta-objeto gerenciador. De qualquer forma, independentemente desta funcionalidade, a questão da análise do tempo de execução em ambientes reflexivos não está totalmente estabelecida e segundo [Mitchell 97] deverá ser assunto de pesquisa durante muito tempo ainda. No caso particular do modelo RTR, esta questão deve ser solucionada de forma particular em cada implementação do modelo.

III.5.7 - Mudança dinâmica do algoritmo de escalonamento

Além da proposta básica do modelo RTR que prevê a realização de um meta-escalonamento a nível de aplicação, permitindo que o projetista defina estaticamente o meta-objeto escalonador a ser usado na aplicação, o modelo RTR também suporta a mudança da política de escalonamento durante a execução da aplicação.

Para tanto, o meta-objeto escalonador deve ser programado de forma a suportar duas ou mais políticas de escalonamento (através da existência de dois ou mais métodos "Escalona()") e uma função auxiliar capaz de analisar o comportamento da aplicação durante a vigência de uma determinada política de escalonamento e, em função desta análise, chavear o escalonador para outra política disponível. Esta análise poderá ser realizada em função de uma qualidade desejada (*default* ou especificada pelo usuário no *start-up* da aplicação), com base, por exemplo, no percentual de violações temporais ocorridas em um determinado intervalo de tempo. Adicionalmente, caso nenhuma das políticas disponíveis satisfaça o padrão de qualidade desejado, poderá ser usada aquela que tenha produzido um melhor resultado.

A mudança de política de escalonamento durante a execução do sistema, é particularmente atrativa para STR que apresentam diferentes modos de funcionamento; neste caso, diferentes fases do sistema podem ser escalonadas segundo diferentes políticas de escalonamento. Além disto, este esquema é apropriado para a fase de testes do sistema, onde diferentes políticas de escalonamento podem ser testadas e a política a ser usada na operação do sistema pode ser escolhida de forma menos intuitiva e melhor fundamentada.

III.5.8 - Uso simultâneo de diferentes políticas de escalonamento

A escolha de uma política de escalonamento adequada é fundamental para que uma aplicação tenha suas restrições temporais satisfeitas. Entretanto, como nem sempre uma única

política é completamente adequada para todas as classes de tarefas tempo real que compõem uma aplicação, o uso simultâneo de diferentes políticas para o escalonamento de diferentes classes de tarefas (segundo suas características temporais e/ou de acordo com diferentes níveis de garantia desejados) pode ser uma boa solução.

Esta abordagem tem sido usada na prática de programação tempo real em vários trabalhos, como por exemplo:

- na linguagem RTCC [Gehani 91], que usa 5 diferentes políticas para escalonar 5 classes distintas de tarefas tempo real (classificadas em função do nível de garantia suportado);

- na linguagem FLEX [Lin 91], que utiliza diferentes políticas no escalonamento de tarefas obrigatórias (*rate-monotonic*) e opcionais (EDF);

- e no RT-MOP (Real-Time Meta-Object Protocol) [Mitchell 97], o qual permite que diferentes grupos de escalonamento implementem diferentes políticas de escalonamento, visando a obtenção de diferentes níveis de garantia.

No modelo RTR, esta diversidade de políticas pode ser obtida através do meta-objeto escalonador, o qual pode ser programado de forma a suportar simultaneamente diversas políticas. Para tanto, o método "Escalona()" deverá ser sobrecarregado através de uma implementação para cada política desejada e diferentes filas de escalonamento deverão ser utilizadas. A política a ser usada para o escalonamento de um método particular poderá ser deduzida dinamicamente pelo MOE ou mesmo pelo MOG em função das características temporais ou de outros atributos tais como prioridade ou categoria do método.

Alternativamente, esta funcionalidade poderia ser obtida através da existência de vários MOE's, cada um implementando uma política distinta de escalonamento (como proposto em [Mitchell 97]); entretanto, isto implica na necessidade de um mecanismo de integração entre os diferentes MOE's para gerenciamento global do escalonamento do sistema, caracterizando assim uma extensão estrutural e semântica do modelo proposto.

A vantagem do modelo RTR sobre RTCC e FLEX é a flexibilidade (derivada do uso de reflexão) na definição das políticas a serem utilizadas (independente do suporte de execução e do ambiente operacional), e na forma pela qual as tarefas são subordinadas a esta ou aquela política. Já com relação ao RT-MOP, o modelo RTR permite adicionalmente que diferentes métodos de um mesmo objeto estejam subordinados a diferentes políticas de escalonamento.

Adicionalmente, no modelo RTR esta funcionalidade pode ser combinada com as funcionalidades introduzidas nas subseções III.5.6 e III.5.7 (análise de escalonabilidade dinâmica e mudança dinâmica da política de escalonamento), resultando em esquemas de escalonamentos extremamente flexíveis e adaptáveis, capazes de contribuir efetivamente para o incremento do grau de satisfação das restrições temporais das aplicações.

III.5.9 - Controlando reflexivamente a disponibilidade de memória

Embora o controle do tempo de processamento seja de importância fundamental para sistemas tempo real, a disponibilidade de tempo não é suficiente para que os requisitos temporais de uma aplicação sejam satisfeitos. É necessário que os demais recursos utilizados, entre eles o recurso memória, também estejam disponíveis no tempo e na quantidade desejados.

Apesar da importância do controle da disponibilidade de memória, grande parte da teoria tempo real (e da teoria de escalonamento em particular) não considera explicitamente este aspecto. Em parte, isto deve-se ao fato de que na prática a memória necessária para uma aplicação é alocada estaticamente (alocação dinâmica é normalmente proibida em função da exigência de previsibilidade) e portanto não há necessidade de controle dinâmico deste recurso, já que o sistema só será colocado em execução se houver disponibilidade de memória, o que pode ser definido em tempo de compilação.

Entretanto, existem muitas aplicações tempo real (dinâmicas ou projetadas para serem executadas em ambientes de propósito geral) onde a garantia estática relativa a disponibilidade de memória não pode ser obtida e portanto a disponibilidade ou não de memória deve ser deduzida dinamicamente. Gerenciadores de memória automáticos (coletores de lixo) embora realizem esta tarefa, normalmente são imprevisíveis e não servem para o propósito de tempo real, havendo portanto a necessidade de gerenciadores especialmente projetados para tempo real (como é o caso, por exemplo do coletor de lixo de RT-Java [Nilsen 96b], o qual possibilita a previsão do tempo necessário para alocação de uma determinada quantidade de memória).

Contudo, a existência de um coletor de lixo previsível, por si só não basta; é necessário um esquema especial que permita a detecção de falta de memória, antes que a tarefa seja escalonada. Em RT-Java, por exemplo, este controle é realizado na etapa de configuração, onde uma nova atividade tempo real só será admitida no sistema se houver garantia de disponibilidade da quantidade de memória necessária.

No modelo RTR, este controle pode ser feito dinamicamente usando a capacidade reflexiva do modelo; ou seja a operação de criação de um objeto tempo real pode ser considerada uma operação reflexiva, a qual só será executada se houver disponibilidade de memória para alocação do objeto que está sendo criado. Esta condição deve ser verificada reflexivamente, e se for o caso um coletor de lixo deve ser ativado para obtenção da quantidade de memória necessária. Naturalmente, para que a operação de criação de objetos mantenha-se previsível o coletor de lixo usado deve ser previsível e o tempo possivelmente gasto na coleta de lixo deve ser levado em conta.

III.6 - Expressando e implementando restrições de sincronização multimídia

As características básicas do modelo RTR fazem com que ele seja particularmente atrativo para modelagem e programação de aplicações tempo real *soft*, incluindo aplicações tempo real abertas [Stankovic 96]. Neste contexto, destaca-se a classe de aplicações multimídia (tele-conferências, ensino a distância, instrutores automáticos, entretenimento (jogos), aplicações médicas, comércio eletrônico, etc.), cujos aspectos temporais podem ser adequadamente tratados através de abordagens do tipo melhor esforço.

Os aspectos temporais inerentes a aplicações multimídia estão diretamente relacionados com a questão de sincronização intramídia (relações temporais entre unidades de uma mídia dependente de tempo) e intermídias (relações temporais entre mídias distintas) [Blakowski 96].

A questão de sincronização em sistemas multimídia é bastante complexa e apresenta inúmeras facetas (vide [Blakowski 96] para um *survey* completo sobre sincronização

multimídia). Neste trabalho, entretanto, estamos particularmente interessados na questão da especificação da sincronização e em como esta especificação pode ser realizada segundo a filosofia do modelo RTR.

III.6.1 - Relações de sincronização multimídia

Para representar as várias situações de sincronização encontradas em aplicações multimídia, adotaremos o modelo de especificação baseado em intervalos [Allen 83]. Com base neste modelo demonstraremos, a seguir, a potencialidade do modelo RTR para representar estas situações.

Segundo o modelo baseado em intervalos, podem ser representados 13 diferentes tipos de relações temporais [Allen 83], as quais (com exceção das inversas) são apresentadas na figura 3.8. Por sua vez, o modelo baseado em intervalos estendido [Wahl 94] consiste de 29 relações de intervalo (definidas a partir de disjunções das 13 relações básicas) que são consideradas relevantes para apresentação multimídia. Estas relações podem ser representadas, de forma simplificada, pelos 10 operadores apresentados na figura 3.9.

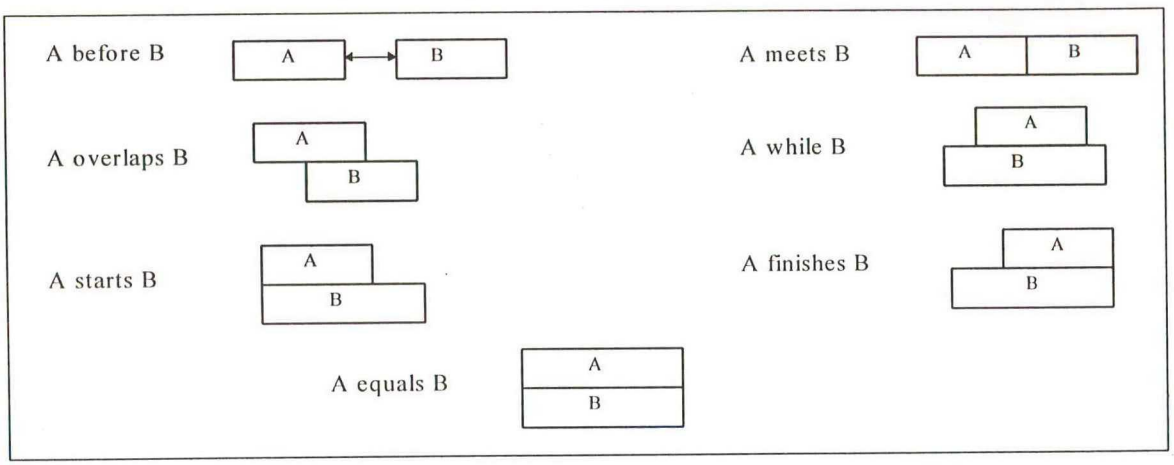


Figura 3.8 - Relações de intervalo básicas

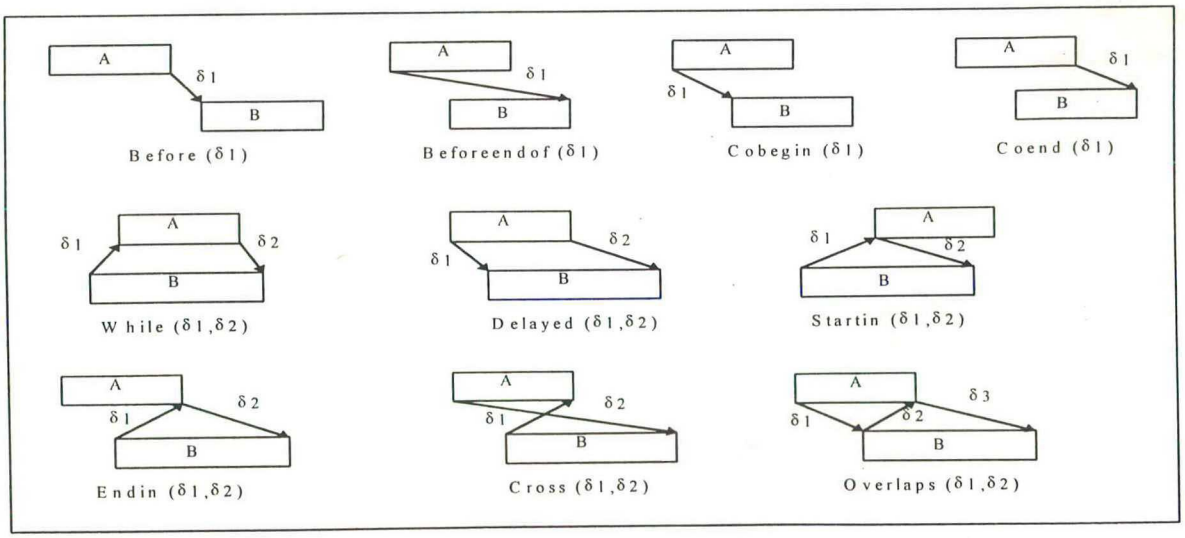


Figura 3.9 - Operadores do modelo baseado em relações de intervalos estendido

No modelo RTR estes operadores (e conseqüentemente as relações de sincronização que eles representam) podem ser facilmente representados através de restrições temporais básicas, tais como "Periodic", "Aperiodic", "Start-at", e "ActivationInterval". Por exemplo, para representar uma animação parcialmente comentada por uma seqüência de áudio, a relação de sincronização descrita em [Blakowski 96] como

Animation **while (d1, d2)** Audio

pode ser expressa no modelo RTR pela associação da restrição temporal "ActivationInterval" aos métodos responsáveis pela apresentação das mídias envolvidas, como mostrado abaixo.

```
void Animation ( ... ), ActivationInterval (StartTime, EndTime),
                                AnimationException ( )
begin ... end;
void Audio ( ... ), ActivationInterval (StartTime, EndTime),
                                AudioException ( )
begin ... end;
```

A sincronização desejada, representada graficamente na figura 3.10, será alcançada pela ativação assíncrona (denotada pelo símbolo "@") dos métodos "Animation()" e "Audio()" abaixo especificada, onde T1 , T2, δ1 e δ2 são atributos temporais da relação de sincronização representada na figura 3.10.

```
...
@Animation ( ... ), (T1, T2);
@Audio ( ... ), (T1 + δ1, T2 - δ2);
...
```

Da mesma forma que o operador while, todos os demais operadores apresentados na figura 3.9, podem ser representados no modelo RTR através de restrições temporais básicas. No apêndice B é apresentada uma das possíveis representações de cada um desses operadores.

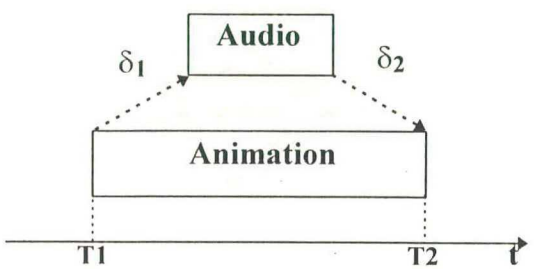


Figura 3.10 - Representação gráfica da relação "Animation while(δ1, δ2) Audio"

Um dos problemas básicos do método de especificação baseado em intervalos, é que ele não permite que as relações temporais entre sub-unidades de uma mídia sejam representadas diretamente (a menos que ela seja explicitamente subdividida). No exemplo apresentado é omitida a relação temporal existente entre os quadros da animação e entre os pacotes de áudio; usando o modelo RTR estas relações podem ser representadas através de métodos periódicos (onde cada período corresponde a um intervalo de apresentação) responsáveis pela apresentação (*display*) das sub-unidades que compõem cada mídia. Assim, a

função básica dos métodos "Animation()" e "Audio()", além de fixarem os intervalos de apresentação das mídias e a relação entre eles, é ativar os métodos periódicos "DisplayVideo()" e "DisplayAudio()" responsáveis pela apresentação dos quadros de imagem e pacotes de som que compõem as mídias consideradas.

Enquanto a restrição de periodicidade garante por si só a sincronização intramídia, ela deve ser redefinida para que a sincronização intermídia possa ser obtida; isto pode, por exemplo, ser obtido através da correlação temporal entre apresentações de sub-unidades de múltiplas mídias, tomando-se uma delas como sendo a mídia mestra. Por exemplo, no caso de sincronização de lábios (*lip-synchronization*), onde um atraso do vídeo com relação ao áudio e vice-versa é tolerável (dentro de limites pré-estabelecidos), o controle de sincronização pode ser efetivado tomando-se a mídia "áudio" como mestra e verificando se a apresentação de cada quadro de "vídeo" está sendo apresentado dentro das restrições toleradas. Este controle pode ser embutido em uma restrição *periodic* convencional, ou por questão de clareza e flexibilidade, pode ser implementado em um novo tipo de restrição temporal que englobe o controle de periodicidade com o controle da sincronização intermídia, como exemplificado abaixo.

```
...
void DisplayAudio ( ... ), Periodic (P=30, MET = 3, Fim),
                              AudioException ( )
begin ... end;
void DisplayVideo ( ... ), AudioSynchronization (P=40, MET=5, Variacao=5,
                                                 VBA=15, VAA=150, Fim),
                              VideoException ( )
begin ... end;
...
```

Segundo este exemplo, a mídia áudio deve ser apresentada a uma taxa de 1 pacote a cada 30ms e não suporta nenhum *jitter*, enquanto que a mídia vídeo deve ser apresentada a uma taxa de 1 quadro a cada 40ms suportando um *jitter* de +/- 5ms, o que implica que um quadro será apresentado entre 35 e 45ms após o quadro anterior. Além disso, o exemplo estabelece que o vídeo pode preceder o áudio associado em até 15ms (atributo "VBA" - Vídeo Before Áudio) e que o vídeo não pode estar atrasado com relação ao áudio correspondente em mais de 150ms (atributo "VAA" - Vídeo After Áudio).

As restrições temporais utilizadas no exemplo apresentado, estabelecem que a apresentação de cada mídia iniciará imediatamente após a ativação dos métodos envolvidos ("DisplayVideo()" e "DisplayAudio()") e se estenderá até o tempo especificado através do atributo "Fim"; alternativamente, o início da apresentação poderia ser explicitamente especificado através de um atributo adicional "Inicio". Além disso, visando flexibilidade e reuso, os valores dos atributos temporais podem ser estabelecidos no momento da ativação dos métodos e não no momento da declaração como mostrado no exemplo utilizado.

III.6.2 - Exemplo de aplicação

Para ilustrar a aplicabilidade do modelo RTR na programação de aplicações multimídia, apresentaremos aqui uma versão simplificada do exemplo utilizado em [Little 90] com o objetivo de exemplificar seu modelo de especificação formal (baseado em OCPN - Object Composition Petri Net) para sincronização, recuperação e apresentação de documentos multimídia.

O referido exemplo, denominado "Instrutor de anatomia e fisiologia" é uma aplicação baseada em híper-mídia e na especificação de restrições temporais baseadas em intervalos. Nesta aplicação, o usuário pode selecionar a lição (unidade de informação) desejada a partir de uma base de dados multimídia existente. Cada lição possui um documento multimídia correspondente contendo além dos dados propriamente ditos, informações referentes a sincronização destes dados e a apresentação do documento.

Por questão de objetividade, mas sem perda de generalidade, nos abstrairemos das questões relativas ao armazenamento/recuperação dos dados multimídia e das informações relativas a sua sincronização e apresentação, detendo-nos especificamente na questão da programação da apresentação de uma lição a partir do conhecimento das relações de sincronização entre as mídias que a compõem. Para exemplificar o uso do modelo nesta aplicação, usaremos uma lição específica ("Músculo do coração"), a qual é detalhadamente apresentada em [Little 90].

O documento multimídia referente a lição escolhida é composto por áudio, vídeo, textos, imagens e animações, os quais são apresentados em uma única tela dividida em regiões (figura 3.11, adaptada de [Little 90]), segundo as relações de sincronização representadas graficamente na figura 3.12. A especificação destas relações usando o modelo baseado em intervalos é mostrada na figura 3.13.

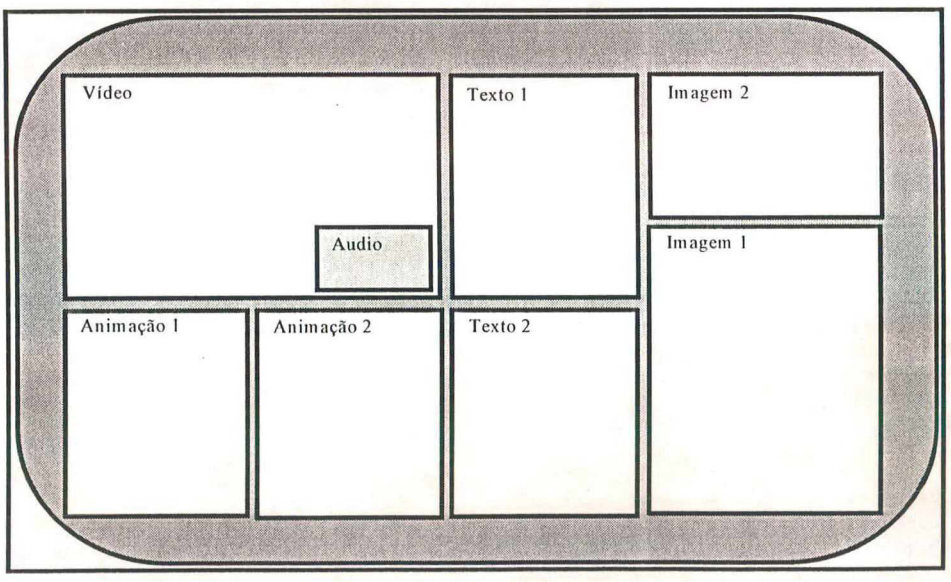


Figura 3.11 - Tela de apresentação da aplicação Instrutor de Anatomia e Fisiologia

Usando o modelo RTR, uma solução para a aplicação em questão pode ser estruturada e programada como mostrado, respectivamente nas figuras 3.14, 3.15 e 3.16. Nesta solução, o OBTR "Controlador" é responsável diretamente pelo controle da sincronização desejada (a nível de mídias), através da solicitação de operações de apresentação enviadas ao OBTR "Apresentador" de acordo com as relações de sincronização especificadas (figuras 3.12 e 3.13). Por outro lado, o controle de sincronização envolvendo as sub-unidades das mídias e a apresentação das mídias propriamente ditas, ficam sob a responsabilidade do OBTR "apresentador".

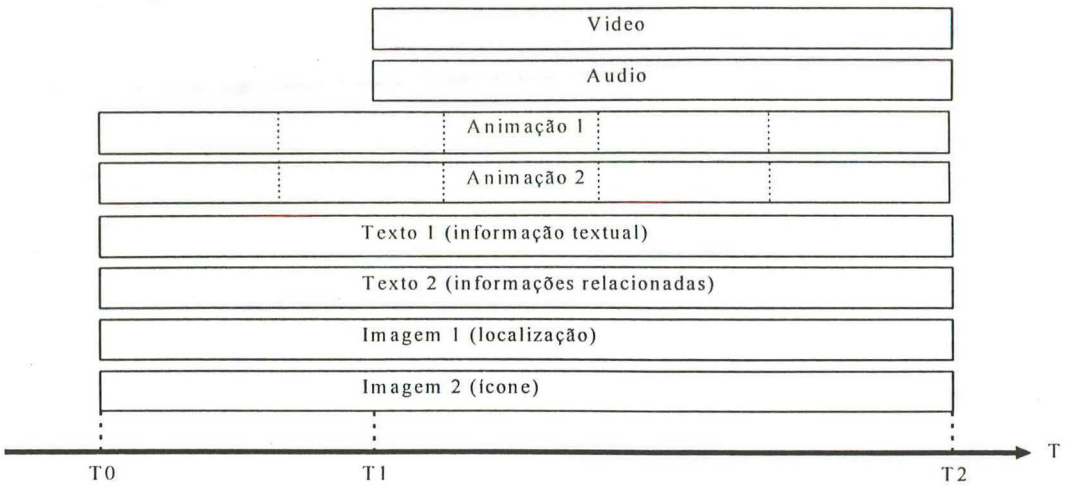


Figura 3.12 - Representação gráfica da sincronização entre as mídias que compõem a lição Músculo do Coração

Animação1 *equals* Animação2
Animação1 *equals* Texto1
Texto1 *equals* Texto2
Texto2 *equals* Imagem1
Imagem1 *equals* Imagem2
Audio *while (T1-T0, 0)* Animação1
Video *while (0,0)* Audio

Figura 3.13 - Representação da sincronização segundo o modelo baseado em intervalos

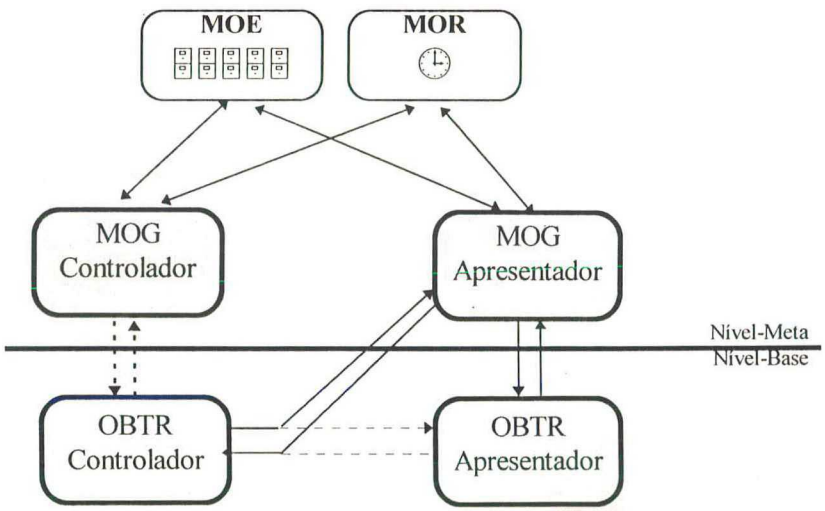


Figura 3.14 - Estrutura da solução proposta

Considerações sobre a solução proposta - Visando um melhor entendimento da solução da aplicação em questão através do modelo RTR, apresentaremos a seguir uma série de considerações relativas a solução proposta:

```
OBTR class Controlador
begin
    ...
    void ApresentaVideo( ... ), ActivationInterval (Tinicio, Tfim), ExceçãoAV( )
    begin
        ...
        Apresentador.DisplayVideo ( ... ), (40, Tfim, 5, 15, 150);
        ...
    end;
    void ApresentaAudio( ... ), ActivationInterval (Tinicio, Tfim), ExceçãoAA( )
    begin
        ...
        Apresentador.DisplayAudio ( ... ), (30, Tfim);
        ...
    end;
    void Animacao( ... ), Periodic (P, Fim=T2, MET=100), ExceçãoAnimacao( )
    begin
        ...
        Apresentador.DisplayImagem ( ... ) (50);
        ...
    end;
    void Main ( )
    begin
        ...
        @ApresentaVideo ( ... ), (T1, T2);
        @ApresentaAudio ( ... ), (T1, T2);
        @Animacao (Animação1, ... ), ( (T2-T0)/5);
        @Animacao (Animação2, ... ), ( (T2-T0)/5);
        @Apresentador.DisplayTexto (Texto1, ...), (100);
        @Apresentador.DisplayTexto (Texto2, ...), (100);
        @Apresentador.DisplayImagem (Imagem1, ...), (100);
        @Apresentador.DisplayImagem (Imagem2, ...), (100);
        ...
    end;
end;
```

Figura 3.15 - Pseudo-código do OBTR "Controlador"

• *Considerações sobre o OBTR "Controlador"*

1 - As relações de sincronização especificadas na figura 3.12, são representadas através da associação de restrições temporais convencionais aos métodos dos objetos-base da aplicação. Por exemplo, a relação "while(0,0)" entre vídeo e áudio é representada pela restrição *ActivationInterval,* onde o uso de valores idênticos para os atributos Inicio e Fim (respectivamente T1 e T2), resulta no comportamento especificado pela relação considerada;

2 - Para obtenção do efeito correspondente as relações *while* e *equal*, as ações de apresentação de mídias são disparadas assincronamente (indicado pelo símbolo "@" na figura

3.15), o que implicará (a nível de implementação) no uso de *threads* distintas para apresentação das diferentes mídias consideradas;

3 - Como pode ser notado, as restrições temporais usadas, apresentam valores *default* para alguns de seus atributos (estabelecidos no momento em que a restrição é associada a um método); contudo, tais valores podem ser substituídos (se a aplicação assim exigir) no momento da ativação dos métodos aos quais as restrições estão associadas.

• *Considerações sobre o OBTR "Apresentador"*

1 - Além das restrições temporais decorrentes das relações de sincronização especificadas, outras restrições temporais foram introduzidas para, por exemplo, representar o *deadline* da apresentação dos textos e imagens (restrição *aperiodic*) e a sincronização intramídia e intermídia relativa as mídias de áudio e vídeo (restrições *Periodic* e *AudioVideoSynchronization*);

2 - No caso da apresentação de textos e imagens, uma implementação fiel a representação utilizada poderia ser a associação da restrição *ActivationInterval* aos métodos "DisplayTexto()" e "DisplayImagem()" do objeto "Apresentador"; entretanto, optamos pelo uso da restrição *Aperiodic* com o objetivo de fazer com que os textos e imagens sejam apresentados no início do intervalo [T0, T2]. Por outro lado, caso não haja necessidade de sincronização de lábios entre o áudio e o vídeo que compõem o documento, o método "DisplayVideo()" poderia ser um método periódico convencional;

3 - Os manipuladores de exceções temporais estão implicitamente estruturados; assim sendo, por exemplo, uma violação de *deadline* na apresentação de uma imagem (método "DisplayImagem()" do objeto "Apresentador" poderá ser tratada localmente ou ser repassada para o objeto "Controlador". Alternativamente, esta relação pode ser explicitada através da associação da cláusula *Timeout* à ativação do método "DisplayImagem()";

4 - Na prática, o objeto "Apresentador" poderia ser sub-dividido em vários objetos de mídia (dependendo da estrutura de armazenamento do documento) e ou delegar funções a objetos especializados (internos ou não) na sincronização e/ou apresentação de determinadas mídias.

```
OBTR class Apresentador
begin
    ...
    void DisplayVideo ( ... ), AudioVideoSynchronization (P, EndTime,
                                Var, VBA, VAA, MET=5), ExceçãoDV ( )
    begin ... end;
    void DisplayAudio ( ... ), Periodic (P, Fim, MET=3), ExceçãoDA ( )
    begin ... end;
    void DisplayTexto ( ... ), Aperiodic (D, MET=10), ExceçãoDT ( )
    begin ... end;
    void DisplayImagem ( ... ), Aperiodic (D, MET=15), ExceçãoDI ( )
    begin ... end;
     ...
end;
```

Figura 3.16 - Pseudo-código do OBTR "Apresentador"

• *Outras considerações*

1 - A forma como as relações de sincronização foram representadas na solução proposta, não é a única possível; diferentes combinações envolvendo diferentes restrições temporais básicas podem ser alternativamente utilizadas;

2 - As restrições temporais assim como os manipuladores de exceções utilizados na definição dos OBTR "Controlador" e "Apresentador", devem ser implementados nos meta-objetos correspondentes. Note-se que com exceção da restrição *AudioVideoSynchronization* (definida especificamente para aplicações multimídia), as demais restrições utilizadas são básicas e podem ser reutilizadas a partir de uma biblioteca de restrições disponíveis em uma implementação particular do modelo;

3 - O meta-objeto escalonador e o meta-objeto relógio a serem usados na presente aplicação, podem ser meta-objetos padrões disponíveis, já que nenhuma funcionalidade adicional é necessária; por exemplo o MOE poderia usar a política de escalonamento EDF (*Earliest Deadline First).* Alternativamente, entretanto, nada impede que um novo MOE usando uma abordagem de escalonamento mais elaborada seja definido pelo usuário;

4 - O exemplo de aplicação apresentado, embora seja um exemplo simples, é bastante significativo na medida em que pode ser usado como base para a solução de aplicações multimídia mais complexas. Complementarmente, deve ser ressaltado que além das relações de sincronização presentes no exemplo considerado, outras relações incluindo todos os operadores de intervalo e interação com o usuário, podem ser similarmente representados, o que possibilitará, por exemplo a programação de aplicações multimídia interativa tais como vídeo-games, trabalhos cooperativos etc.

III.7 - Uma extensão do modelo RTR para ambientes distribuídos abertos

A integração de tempo-real e distribuição é atualmente alvo de intensas pesquisas, quer através de projetos isolados como por exemplo [Takashio 92], [Kim 94a, 94b] e [Mitchel 97], quer através de projetos de grande porte como é o caso dos esforços despendidos no âmbito dos projetos ANSA [Li 94a, 94b] e CORBA-RT [OMG 96b]; este esforço de pesquisa visa satisfazer um dos requisitos básicos dos atuais e futuros STR que cada vez mais são distribuídos.

Nesta seção, descreveremos uma proposta de extensão do modelo RTR para ambientes distribuídos abertos, baseado na arquitetura CORBA. Além de apresentarmos os fundamentos básicos da proposta, também apresentamos um exemplo de uma aplicação multimídia distribuída; encerrando a seção descrevemos um protótipo da extensão proposta realizado sobre a plataforma Solaris (TM Sun Microsystems Inc.). A descrição da especificação e da implementação desta extensão foi objeto de uma dissertação de mestrado do LCMI [Siqueira 96] e de diversos artigos nacionais e internacionais [Fraga 95, 96, 97a], [Furtado 96a].

III.7.1 - Aspectos básicos da extensão proposta

A integração de tempo real em ambientes distribuídos abertos, mais precisamente em sistemas distribuídos de larga escala, deve levar em consideração os seguintes aspectos [Takashio 92, 96] :

- O fato de que em tais ambientes o tempo de comunicação não é limitado, implicando na adoção de estimativas para o atraso máximo no suporte de comunicação;

- A impossibilidade de se ter relógios locais sincronizados dentro de um nível de granularidade desejado, inviabilizando a verificação de restrições temporais em chamadas remotas com base em um tempo global;

- O fato de que a carga em cada nodo do sistema pode ser dinâmica e imprevisível, podendo levar a não satisfação das restrições temporais dos serviços solicitados.

Em função destes aspectos, a proposta de integração de tempo real em ambientes abertos, deve levar em conta as seguintes premissas:

- A falta de previsibilidade (local e global) inviabiliza o fornecimento de garantias, impondo a adoção de políticas de melhor esforço (best-effort) para o escalonamento das tarefas do sistema;

- A verificação da satisfação das restrições temporais deve ser feita com base em relógios locais;

- No caso de adoção do modelo de interação cliente/servidor, o cliente deve estabelecer um valor de *Timeout* com base em suas restrições temporais, o qual deve ser um valor estimado, uma vez que neste tipo de sistema é impossível realizar uma análise de pior caso. A partir do *timeout* especificado e de uma estimativa do tempo de comunicação, deve ser obtido um valor de *Deadline,* que limitará o tempo no qual o método requisitado deve ser executado no servidor;

- As tarefas deverão primeiro ser alocadas aos nodos do sistema para posteriormente serem escalonadas segundo critérios locais [Burns 90].

Com base nas premissas acima especificadas, definimos uma extensão distribuída do modelo RTR para ambientes abertos, a qual consiste em adaptar o modelo RTR básico a arquitetura CORBA, com o objetivo de prover interoperabilidade entre componentes heterogêneos distribuídos.

Na extensão proposta, aplicações distribuídas são organizadas segundo o modelo cliente-servidor, sendo que tanto clientes quanto servidores são estruturados em nível-base e nível-meta de acordo com a estrutura básica do modelo RTR. Em cada nodo onde a aplicação estiver sendo executada, haverá também um meta-objeto relógio (MOR) correspondente ao relógio local, e um meta-objeto escalonador (MOE), responsável pelo escalonamento das tarefas locais. Além disso, para encapsular interfaces CORBA e manipular comunicação remota, serão adicionados meta-objetos de comunicação ao modelo RTR.

III.7.2 - Arquitetura CORBA

CORBA (*Common Object Request Broker Architecture*) é um conjunto de padrões e conceitos propostos pela OMG (*Object Management Group*) para sistemas abertos. Na arquitetura proposta (figura 3.17), requisições de métodos de objetos são feitas transparentemente em um ambiente distribuído e heterogêneo através do ORB (*Object Request Broker*).

A especificação CORBA [OMG 96a] define as interfaces do ORB e estabelece o papel de cada um de seus componentes no ambiente distribuído. As interfaces CORBA são especificadas como uma camada, mascarando diferenças entre as diversas implementações possíveis.

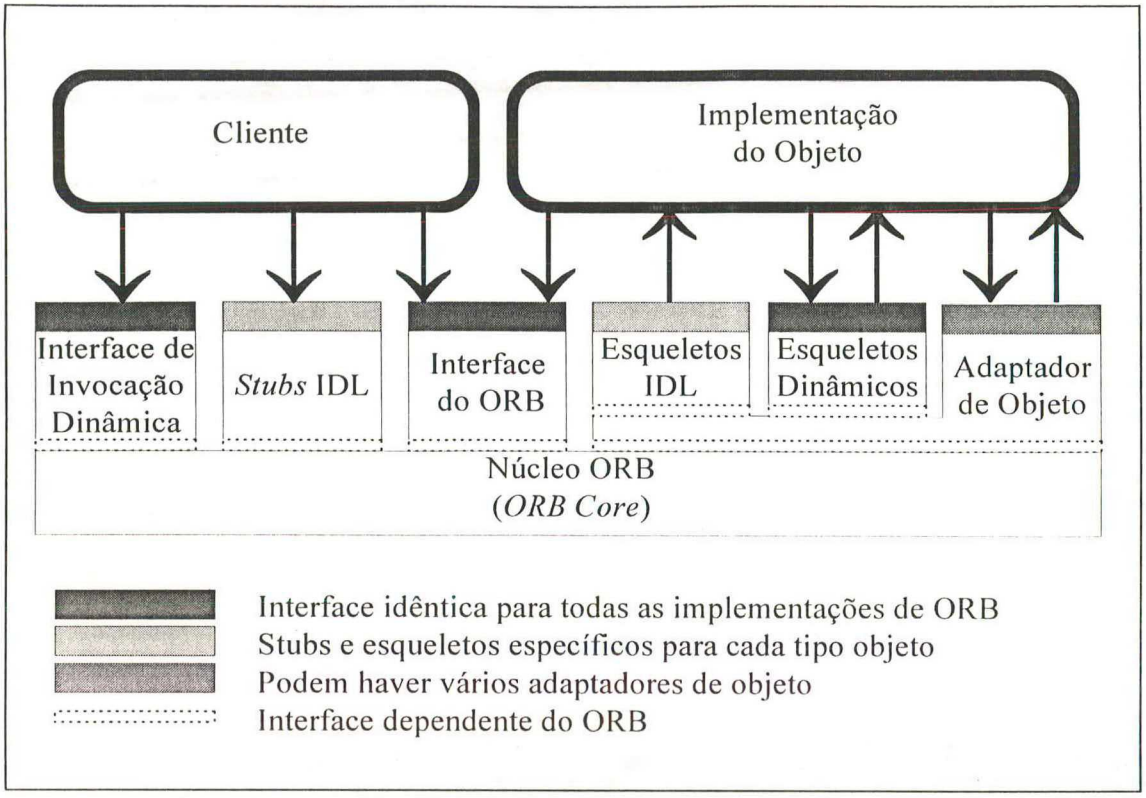


Figura 3.17 – Arquitetura CORBA

Em um ambiente CORBA, cada objeto tem sua interface especificada na linguagem IDL (*Interface Definition Language*), uma linguagem puramente declarativa com sintaxe e tipos predefinidos baseados na linguagem C++. Para requisitar a execução de um método, um cliente CORBA utiliza *stubs* geradas na compilação da especificação de interface IDL do objeto servidor, ou pode construir uma requisição utilizando a interface de invocação dinâmica DII (*Dynamic Invocation Interface*). Para permitir invocações dinâmicas, interfaces de objetos devem ser armazenadas em um depósito de interfaces (*Interface Repository*).

O ORB implementa semânticas e abstrações da comunicação entre objetos na rede; ele transmite a requisição através da rede e transfere o controle para o adaptador de objetos utilizado para ativar a operação na implementação do objeto (ou seja, no servidor) através de *skeletons* IDL estáticos ou dinâmicos. A implementação de objeto (servidor) trata a requisição da mesma forma, independentemente do mecanismo de invocação de método utilizado – estático ou dinâmico.

III.7.3 - Objetos-Base e Meta-Objetos de comunicação

Seguindo a abordagem RTR, as interfaces CORBA (*stubs*, DII e *skeletons*) usadas na comunicação remota devem ser transparentes aos objetos-base. Para tanto, aspectos referentes à comunicação remota entre processos devem ser tratados por meta-objetos adicionais de comunicação, os quais devem encapsular os aspectos referentes a estas interfaces, possibilitando a invocação de métodos remotos de maneira transparente ao programador.

As chamadas de métodos remotos originadas em um objeto-base são redirecionadas para *MetaStubs*, que modificam as *stubs* geradas pelo compilador IDL-CORBA de modo a tratar os requisitos temporais de aplicações tempo-real. MetaStubs são responsáveis por, interagindo com o meta-objeto relógio, controlar o *timeout* das chamadas de métodos do

objeto-base. Como alternativa à chamada por *stubs*, pode ser utilizada a interface de invocação dinâmica (DII) do CORBA e um meta-objeto MetaDII.

As chamadas de métodos originárias de objetos remotos e destinadas a um determinado objeto servidor são recebidas através de MetaSkeletons, que modificam a funcionalidade dos *skeletons* gerados pelo compilador IDL. Estes recebem a requisição através do ORB e chamam o respectivo método no objeto-base. Esta chamada é desviada para o meta-objeto gerenciador correspondente que, interagindo com o meta-objeto escalonador e com o meta-objeto relógio de seu nodo, fará a verificação das restrições temporais associadas aos métodos solicitados.

Em uma aplicação distribuída, um cliente pode utilizar serviços de diferentes servidores; isto implica que um cliente precisa de um MetaStub para cada um de seus servidores. Se a ligação entre cliente e servidor for dinâmica, um MetaDII é suficiente; o cliente estabelece conexões com servidores utilizando a mesma interface DII CORBA. Da mesma forma, servidores podem suportar vários MetaSkeletons.

III.7.4 - Cláusula *timeout*

Para possibilitar o controle temporal no lado cliente de uma interação do tipo cliente/servidor, a ativação de um método do servidor deve possuir uma cláusula *timeout* associada, como especificado na figura 3.18.

```
<Id-Objeto> . <Id-Metodo>(<Argumentos-Funcionais>), [(<Argumentos-Temporais>)]
    timeout (<Valor-Timeout>),
    exception
    begin
        case timeout : <Id-Manipulador-Exceção>
        [case reject  : <Id-Manipulador-Exceção>]
        [case abort   : <Id-Manipulador-Exceção>]
    end
```

Figura 3.18 - Cláusula *timeout*

Esta cláusula estabelece um tempo limite para espera dos resultados do método solicitado. Se o retorno não ocorrer dentro do tempo estabelecido, uma exceção será levantada e o manipulador de exceções correspondente ao *case timeout* será executado. Por outro lado, caso o *timeout* não tenha sido violado mas a restrição temporal associada ao método solicitado seja violada antes ou durante a execução do mesmo, o cliente receberá uma notificação de exceção do tipo "*reject*" ou "*abort*". Nestes casos, a exceção ocorrida será tratada, respectivamente, pelos manipuladores de exceção associados a *case reject* e *case abort*. Os procedimentos de exceção podem consistir na utilização de uma réplica do servidor, na execução de uma versão simplificada do método, ou simplesmente na emissão de uma mensagem de erro.

Contudo, o uso de exceções do tipo *timeout* no lado cliente pode levar a problemas de consistência no sistema, já que quando uma exceção deste tipo é levantada, o cliente não tem como saber se a execução do método solicitado foi ou não concluída, uma vez que a resposta pode ter sido atrasada no suporte de comunicação. Para tratar esta questão foi proposto um mecanismo de sincronização de estado, com base no qual o cliente pode forçar uma nova sincronização na interação cliente servidor. Este mecanismo foi definido a partir da

capacidade do suporte CORBA em sinalizar exceções em servidores e na rede de comunicação.

```
case timeout :
   ...
   waitstate ( <Tempo-Máximo-De-Espera> )
   begin
        case concluded:
                <Id-Manipulador-Exceção>
        case rejected :
                < Id-Manipulador-Exceção >
        case aborted :
                < Id-Manipulador-Exceção >
        case unknown :
                < Id-Manipulador-Exceção >
   end;
   ...
```

Figura 3.19 - Mecanismo para sincronização de estado em uma interação cliente/servidor

O mecanismo proposto, cuja estrutura é mostrada na figura 3.19, permite que após a ocorrência de uma exceção do tipo *timeout* a resposta atrasada possa ser aguardada por um período adicional de tempo (<tempo-máximo-de-espera>). O resultado enviado pelo servidor e recebido no comando *waitstate*, pode indicar que o método foi concluído, rejeitado ou abortado, permitindo uma re-sincronização do estado do sistema através da realização de ações alternativas específicas a cada um destes casos; por outro lado, é possível que nenhuma resposta seja produzida dentro do tempo especificado, neste caso o tratamento de exceção correspondente ao estado "*unknown*" deve ser executado.

III.7.5 - Comportamento da comunicação no modelo proposto

A comunicação no modelo RTR distribuído pode ser síncrona ou assíncrona. No caso síncrono, a chamada de um método do objeto-base servidor deve possuir um *timeout* associado, cujo controle será realizado pelo MetaStub ou pelo MetaDII; por outro lado, no caso assíncrono não haverá controle do *timeout*, já que não há bloqueio do cliente. A seguir, com base na figura 3.20, descrevemos o comportamento dos componentes do modelo no caso de uma chamada síncrona.

Uma chamada de método realizada em um objeto-base cliente é encaminhada via MetaStub para o servidor ao qual se destina a requisição (ação ①). O MetaStub requisita o método chamado pelo objeto-base através do ORB utilizando *stubs* (ação ②), e envia ao meta-objeto relógio o valor de *timeout* associado à chamada (ação ❷). O MetaStub passa a aguardar uma mensagem de resposta do servidor, ou do meta-objeto relógio, sinalizando uma violação de *timeout*. Caso seja utilizada a interface de invocação dinâmica, será utilizado um meta-objeto DII que fará o controle de *timeout* antes de enviar a requisição do método através da interface DII do CORBA.

A requisição do método chega ao servidor correspondente através do ORB, que ativa o método utilizando *skeletons* CORBA e MetaSkeletons. A ativação é desviada para o meta-objeto gerenciador do servidor (ação ③), que interage com o meta-objeto escalonador daquele

nodo (ação ④), aguardando por uma autorização para ativar o método requisitado. Quando o meta-objeto escalonador sinaliza que o método deve ser executado, o meta-objeto gerenciador do servidor verifica se as restrições temporais (ação ⑤) e de sincronização foram violadas, retornando uma exceção `reject` neste caso. Caso contrário, o método é chamado no objeto-base do servidor (ação ⑥). Concluída a chamada, ocorre novamente a verificação das restrições temporais, resultando em uma exceção `abort` caso estas tenham sido violadas. O resultado da chamada é enviado na forma de parâmetros ao objeto-base do cliente, passando através do ORB (usando *stubs* e *skeletons*) e de MetaStub (ações ⑦, ⑧e ⑨).

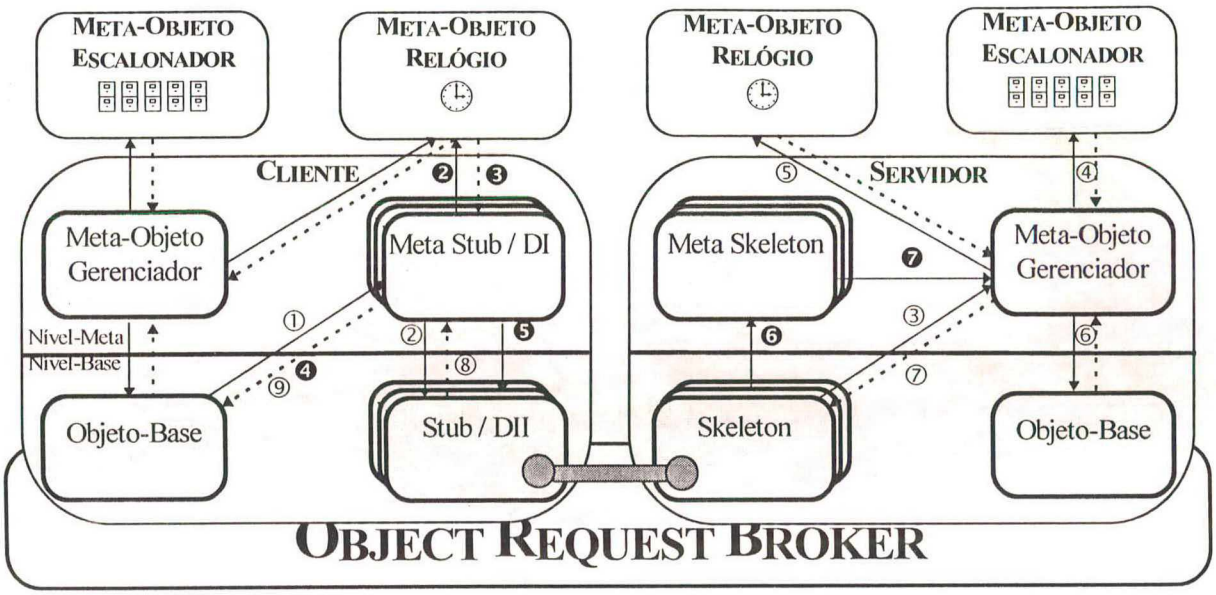


Figura 3.20 – Estrutura da extensão do Modelo RTR para ambientes distribuídos abertos

No caso de, antes da finalização do protocolo, o meta-objeto relógio reportar ao MetaStub uma violação de *timeout* (ação ❸ da Figura 3.20), uma indicação de exceção é enviada ao objeto cliente (ação ❹). Em seguida, o MetaStub remete através do ORB e do MetaSkeleton a indicação de exceção do tipo `timeout` ao meta-objeto gerenciador do servidor (ações ❺, ❻ e ❼), que se encarrega de cancelar a execução do método, retirando a requisição da fila de pedidos do escalonador ou interrompendo a execução do método no objeto-base.

III.7.6 - Um exemplo de aplicação

Para ilustrar a extensão distribuída do modelo RTR proposta, apresentamos a seguir um exemplo de uma aplicação multimídia distribuída. A aplicação escolhida consiste em construir um documento multimídia em uma estação usuária a partir de mídias provenientes de estações fornecedoras, localizadas em diferentes nodos de uma rede [Karmouch 93]. A estrutura da aplicação vista sob a ótica do modelo cliente/servidor é mostrada na figura 3.21.

Dentre os requisitos desta aplicação, destacamos a necessidade de apresentação do documento em tempo real com uma qualidade aceitável e a não interrupção da apresentação, a qual pode no entanto sofrer uma degradação de qualidade dentro de limites aceitáveis. Estes requisitos implicam no estabelecimento e no controle de restrições temporais associadas aos

objetos da aplicação. Por outro lado, não estamos considerando neste exemplo aspectos relativos ao controle de QoS e de sincronização inter-mídias, detendo-nos especificamente nos aspectos relativos a interação cliente/servidor e nas questões temporais envolvidas nesta interação.

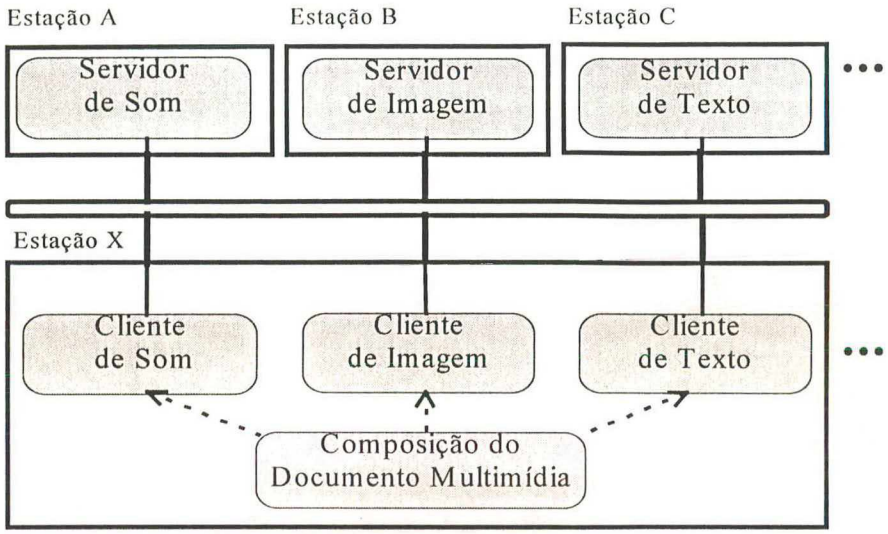


Figura 3.21 – Estrutura da aplicação

Solução proposta - Na solução da aplicação segundo o modelo RTR, clientes e servidores são estruturados na forma de objetos-base e meta-objetos. Para simplificar a descrição da solução proposta, consideraremos uma mídia genérica (representando as diversas mídias envolvidas). No que segue descrevemos as funcionalidades básicas dos clientes e servidores de mídia e esquematizamos sua representação (objetos-base e meta-objetos gerenciadores) segundo o modelo RTR.

Cliente de mídia - É responsável pela obtenção e apresentação de uma mídia específica. A requisição e a apresentação da mídia são realizadas através de métodos periódicos (métodos "RequisitaMidia()" e "ApresentaMidia()" da figura 3.22), sendo que a solicitação da mídia ao objeto servidor é controlada por um *Timeout*, cujo valor é estabelecido em função da taxa (velocidade) de apresentação e da capacidade de armazenamento do objeto em questão. Para evitar descontinuidade na apresentação da mídia, serão usados dois *buffers*; assim, enquanto o conteúdo de um *buffer* está sendo apresentado o outro poderá estar sendo preenchido. O objeto-base (OBTR) e o meta-objeto gerenciador (MOG) referentes a um cliente de mídia são esquematizados na figura 3.22.

Servidor de mídia - Atua em um esquema de fornecimento de mídia sob demanda; ou seja, os dados são enviados ao cliente quando requisitados, segundo o fluxo de execução e a taxa de transmissão impostos pelo cliente. Assim sendo, a operação de fornecimento de dados (método RecuperaMidia(), na figura 3.23) é vista como sendo uma operação aperiódica, cujo *deadline* especifica o tempo máximo dentro do qual a operação deve ser executada; o valor deste *deadline* é estimado a partir do tempo máximo de comunicação previsto e do *timeout* estabelecido pelo cliente na ativação do método. O objeto-base (OBTR) e o meta-objeto gerenciador (MOG) referentes a um servidor de mídia são esquematizados na figura 3.23.

```
OBTR Class ClienteMídia
begin
    ...
    RequisitaMídia (...), Periodic (StartTime, Period, EndTime , MET),
                        ExceçãoRequisitaMídia ()
    begin
        ...
        ServidorDeMídia.RecuperaMídia(Buffer[I], ...), (ValorDeadline)
        Timeout (ValorTimeout),
        Exception
            begin
                case reject   : < manipulador de exceção reject >
                case abort    : < manipulador de exceção abort >
                case timeout : < manipulador de exceção timeout >
            end
        ...
    end
    ApresentaMidia (...), Periodic (StartTime, Period, EndTime , MET ),
                        ExceçãoApresentaMidia()
    begin ... end;
    // outros métodos do objeto-base
    ...
end;

MOG Class ClienteMídia
begin
    // seção de gerenciamento
    ...
    // seção de sincronização
    ...
    // seção de exceções temporais
    ExceçãoRequisitaMídia ()
    begin ... end;
    ExceçãoApresentaMidia ()
    begin ... end;
    // seção de restrições temporais :
    Periodic (...)   // implementa a restrição de periodicidade >
    begin ... end;
end;
```

Figura 3.22 - Objeto-base e Meta-Objeto ClienteDeMidia

Outros objetos - Além dos objetos-base e dos meta-objetos relativos aos clientes e servidores de mídia, a solução proposta requer os seguintes objetos e meta-objetos:

- Um objeto-base (e seu respectivo meta-objeto gerenciador) responsável pela composição do documento, cuja função será obter informações sobre o documento a ser apresentado (tais como mídias envolvidas, tempo de apresentação e cenário de sincronização) e criar os clientes relativos as mídias envolvidas, de acordo com o cenário especificado.

- Um meta-objeto relógio e um meta-objeto escalonador em cada nodo onde a aplicação irá ser executada; os meta-objetos escalonadores podem por exemplo, utilizar uma política de escalonamento tal como EDF ("Earliest Deadline First").

- "Stubs" e "Skeletons" gerados a partir da especificação IDL dos diversos objetos que compõem a aplicação; a figura 3.24 mostra a especificação IDL do objeto servidor de mídia a partir da qual serão gerados (pelo compilador IDL) o "Skeleton" e o "Stub" deste servidor. Nesta especificação, a possibilidade de ocorrência de exceções temporais durante a execução do método é indicada pela cláusula **`raises`** (`ExceçõesTemporais`), cujo tratamento foi implementado no arquivo incluído "`exceções_temporais.idl`".

- MetaStubs e MetaSkeletons correspondentes aos Stubs e Skeletons gerados. Estes meta-objetos terão a função de modificar o comportamento de seus objetos-base correspondentes, levando em consideração as restrições associadas aos métodos ativados. Por exemplo, o MetaStub correspondente ao servidor de mídia será responsável pelo controle do *timeout* associado a uma ativação do método "RecuperaMidia()".

```
OBTR Class ServidorDeMídia
begin
    ...
    // declaração de métodos
    RecuperaMídia (...), Aperiodic (Deadline, MET=30),
                          ExceçãoRecuperaMídia ()
    begin ... end;
    // Outros métodos do servidor
    ...
end;

MOG Class ServidorDeMídia
begin
    // seção de gerenciamento
    ...
    // seção de sincronização
    ...
    // seção de exceções :
    ExceçãoRecuperaMídia ()
    begin ... end;
    ...
    // seção de restrições temporais :
    Aperiodic (...) // Implementa a restrição de aperiodicidade
    begin ... end;
end;
```

Figura 3.23 – Objeto-base e Meta-objeto ServidorDeMídia

```
#include <exceções_temporais.idl>
...
interface ServeMídia
{
  ...
  void RecuperaMídia ( ... )
      raises ( ExceçõesTemporais );
  ...
};
```

Figura 3.24 – Interface IDL do objeto-base ServidorDeMídia

III.7.7 - Protótipo da extensão proposta

Para testar e validar a extensão distribuída do Modelo RTR, um protótipo foi implementado sobre o sistema operacional *Solaris (TM Sun Microsystems Inc.)*. Este protótipo baseou-se em um mapeamento do modelo RTR para *Solaris* realizado em [Siqueira 96] e foi testado na implementação de uma aplicação multimídia distribuída similar ao exemplo apresentado na seção anterior.

Considerações gerais sobre o mapeamento realizado - O *Solaris* possui características que permitem sua utilização no desenvolvimento de aplicações tempo real; dentre estas características destacamos sua capacidade *multi-threading* e seu mecanismo de escalonamento baseado em prioridades e classes de escalonamento, no qual destaca-se a classe RT, destinada a processos tempo real.

No mapeamento do modelo para *Solaris* feito neste trabalho, optamos pela solução na qual cada par composto por um objeto-base e seu meta-objeto gerenciador, juntamente com os objetos adicionais de comunicação utilizados na interação com outros objetos do modelo – *stubs* e MetaStubs, *skeletons* e MetaSkeletons – corresponde a um processo *Solaris*. Meta-objeto escalonador e meta-objeto relógio são implementados em processos distintos.

Os processos nos quais residem os objetos-base e meta-objetos, utilizaram *threads* para permitir concorrência interna nos meta-objetos, viabilizando assim o processamento simultâneo de várias requisições. As threads utilizadas foram classificadas em dois níveis: *threads-base*, responsáveis pela execução dos métodos dos objetos-base e *meta-threads*, responsáveis pelos procedimentos de nível-meta; as *meta-threads*, por sua vez, foram divididas *em threads de gerenciamento* e *threads de dispatch*. As *threads de gerenciamento* ficaram responsáveis pela interação com o ORB, recebendo e enviando requisições de métodos, e pela criação de uma *thread de dispatch* para cada invocação de método recebida. As *threads de dispatch* ficaram responsáveis pelo processamento das requisições recebidas, tratando-as segundo a dinâmica de funcionamento do modelo RTR básico.

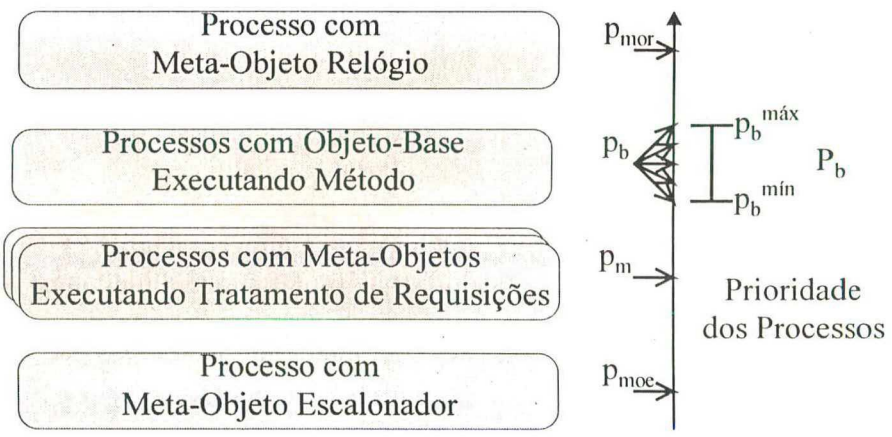


Figura 3.25 – Prioridades de escalonamento para os processos

Para obtenção do comportamento estabelecido pelo modelo RTR, definiu-se um esquema especial de atribuição de prioridades [Siqueira 96] aos processos contendo objetos-base e meta-objetos de uma aplicação tempo real. Segundo este esquema, os processos deveriam pertencer a classe de escalonamento *Real-Time* (RT) *Solaris* e possuírem prioridades variadas, de acordo com a atividade em execução (figura 3.25 [Siqueira 96]).

Conforme representado na figura 3.25, enquanto estiverem processando requisições de métodos a nível-meta, os processos possuirão uma mesma prioridade fixa (Pm). Entretanto, ao receber a liberação do meta-objeto escalonador para executar um método a nível-base, sua prioridade será modificada para um valor entre $Pb^{máx}$ - $Pb^{mín}$, determinado pelo escalonador em função do número de objetos suspensos aguardando o retorno de chamadas síncronas naquele nodo da rede. Da mesma forma, como mostrado na figura 3.26 [Siqueira 96], foi definido um esquema de prioridades para os diferentes tipos de *threads* utilizados.

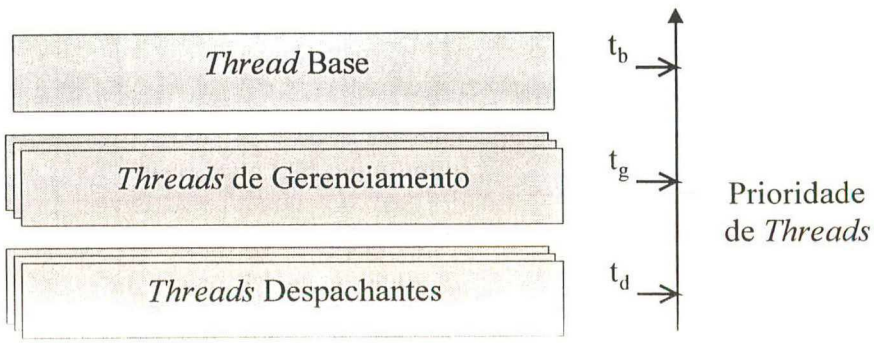


Figura 3.26 – Prioridades de escalonamento para as *threads*

Considerações gerais sobre o protótipo implementado - Na implementação do protótipo do modelo RTR distribuído realizada sobre o sistema operacional *Solaris* versão 2.4, foi utilizado o produto ORBline 1.0 [PMC 95], uma plataforma de suporte das especificações CORBA, de responsabilidade da Post Modern Computing. O ORB utilizado, por não ser *multithreading,* impediu a concorrência interna nos meta-objetos gerenciadores e fez com que os meta-objetos de comunicação fossem implementados através da interface de invocação dinâmica e do depósito de interfaces. Além disso, em função de problemas encontrados na versão utilizada, os valores das restrições e exceções temporais foram passados na forma de parâmetros adicionais dos métodos invocados e não na forma de objetos de contexto CORBA como previsto inicialmente.

Para ilustrar o protótipo, foi desenvolvida uma aplicação multimídia distribuída, similar ao exemplo apresentado na seção anterior. Os objetos-base e meta-objetos usados na aplicação foram programados em C++, e em função da ausência de suporte a reflexão nesta linguagem, o desvio de chamadas de um método de um objeto-base para seu meta-objeto gerenciador correspondente foi programado explicitamente e a associação entre objetos-base e meta-objetos gerenciadores foi obtida através de um apontador para o objeto-base mantido pelo meta-objeto gerenciador. Pelo mesmo motivo, a associação de restrições temporais a declaração e a ativação de métodos, foi realizada através do uso de parâmetros adicionais nestes métodos.

O meta-objeto escalonador implementado utilizou a política EDF (Earliest Deadline First), realizando dinamicamente o mapeamento do *deadline* das tarefas para prioridades (da classe tempo real) do Solaris, de acordo com um esquema previamente definido [Siqueira 96]. Os meta-objetos de comunicação (meta-stubs e meta-skeletons) interagiram com o suporte de comunicação provido pelo ORB, modificando o comportamento de *stubs* e *skeletons* gerados pelo compilador IDL. No protótipo implementado meta-stubs utilizaram a interface de invocação dinâmica para efetuarem uma requisição de um método remoto. Não houve

necessidade de alteração dos *skeletons* IDL, o que implicou na utilização de meta-skeletons nulos.

Considerações finais - O protótipo implementado permitiu uma avaliação preliminar tanto da extensão distribuída como da própria abordagem de programação do modelo RTR. As simplificações e limitações do protótipo com relação a extensão proposta originalmente, foram decorrentes das limitações do ORB utilizado e não da extensão em si. Em função disso, outras implementações da extensão proposta, usando ORBIX e CHORUS-COOL foram realizadas posteriormente [Fritske 97].

Por outro lado, a ausência de reflexão computacional na linguagem utilizada (C++), forçou a simulação de vários aspectos do modelo RTR, reduzindo alguns de seus benefícios. Contudo, esta questão poderá ser superada na medida em que linguagens com suporte a reflexão computacional, tais como Java/RTR (descrita no capítulo subseqüente), puderem estar disponíveis.

Outro aspecto interessante a ser futuramente considerado, é a introdução de facilidades reflexivas no CORBA através de um MOP ("Meta-Object-Protocol"), atualmente em fase de padronização [OMG 96c]. Estas facilidades poderão vir a ser consideradas em futuras implementações da extensão distribuída proposta ou mesmo na proposição de novas extensões.

III.8 - Comparação do modelo RTR com outros modelos

Vários modelos de programação tempo real baseados em objetos e/ou reflexão computacional têm sido propostos nos últimos anos, alguns definidos explicitamente (RTO.k [Kim 94a], DRO [Takashio 92], R^2 [Honda 94], RTT [Gustafsson 94] e o RT-MOP [Mitchell 97]) e outros embutidos implicitamente em linguagens de programação tempo real (Flex [Lin 91], RTC++ [Ishikawa 90, 92] e RT-Java [Nilsen 95, 96a]).

Da mesma forma que o modelo RTR, estes modelos também visam simplificar o projeto e/ou a programação de STR, oferecendo facilidades relativas a estruturação dos sistemas e a representação explicita de restrições temporais. Entretanto, o modelo RTR distingue-se dos demais por reunir características que conjuntamente não são encontradas em nenhum deles; características estas que como será visto a seguir, fazem com que o modelo RTR seja mais flexível e conseqüentemente mais adaptável a diferentes classes de aplicações, sobretudo àquelas que suportam políticas do tipo melhor esforço.

Comparativamente com os modelos RTO.k, Flex, RTC++, RTT e RT-Java, que não são reflexivos, o modelo RTR é no mínimo tão expressivo quanto eles (pois todas as facilidades destes podem ser representadas no modelo RTR), com vantagens adicionais devidas ao uso de reflexão. Em geral, os modelos citados apresentam um conjunto fixo de restrições temporais e políticas de escalonamento, as quais, via de regra, são dependentes do suporte de execução e/ou de ambientes operacionais específicos, impedindo a definição e o uso de novos tipos de restrições temporais e algoritmos de escalonamento a nível da aplicação; aspectos estes que no modelo RTR podem ser livremente escolhidos, definidos e modificados de acordo com as necessidades das aplicações.

Com relação aos modelos DRO, R^2 e o RT-MOP, que como RTR são reflexivos, existem diferenças básicas relativas a filosofia de programação, aos aspectos tratados de

forma reflexiva e a abrangência e efetividade dos modelos propostos; tais diferenças são comentadas nos parágrafos subseqüentes.

DRO é um modelo de objetos tempo real distribuído que suporta as propriedades de melhor esforço e menor prejuízo através de um mecanismo de invocação polimórfica. DRO utiliza meta-objetos para controle de sincronização e para definição de protocolos de comunicação com comportamento tempo-real; aspectos estes que no modelo RTR podem ser tratados, respectivamente, via seção de sincronização e restrições temporais especiais. Por outro lado, o conjunto de restrições temporais de DRO é fixo e não existe a noção de escalonamento a nível de aplicação. Isto eqüivale a dizer que, ao contrário de RTR, as questões relativas a restrições temporais e algoritmos de escalonamento são dependentes do suporte subjacente, o que diminui a flexibilidade do modelo.

R^2 é uma arquitetura reflexiva baseada em objetos, cujo estudo serviu como ponto de partida para a proposta do modelo RTR. Da mesma forma que o modelo RTR, a arquitetura R^2 usa reflexão para definir e controlar restrições temporais e algoritmos de escalonamento a nível de aplicação; entretanto, apesar das similaridades de propósito entre R^2 e RTR, os modelos diferem em vários aspectos, dentre os quais destacamos :

- escopo das restrições temporais - enquanto no modelo RTR as restrições são associadas a métodos, no modelo R^2 elas são associadas a segmentos de códigos; isto, além de ferir a uniformidade do modelo de objetos, dificulta o gerenciamento da complexidade de sistemas tempo real e afeta a capacidade de reuso e manutenção do software produzido com base neste modelo.

- tipos de restrições temporais - Em função do escopo das restrições temporais de R^2, torna-se inviável neste modelo a definição de restrições temporais que viabilizem a ativação de métodos (*script's*) por relógio; já no modelo RTR isto é possível através do uso restrições temporais do tipo "TT" (*Time-Trigger*). Em função disso, restrições temporais como periodicidade não podem ser definidas diretamente, devendo ser simuladas através de delay's embutidos em loop's. Embora no caso geral R^2 flexibilize o uso de diferentes tipos de restrições temporais, a limitação acima mencionada reduz seu poder expressivo.

- abrangência do meta-escalonamento - enquanto no modelo RTR todos os métodos com ou sem restrição temporal são considerados, no modelo R^2 apenas os segmentos de código com restrições associadas são submetidos a tal escalonamento; este esquema dificulta a detecção antecipada de violações temporais, reduzindo a possibilidade de realização de ações alternativas e a efetividade com que as restrições temporais da aplicação podem ser satisfeitas.

- concorrência/sincronização - Enquanto no modelo RTR estas questões são tratados de forma integrada com as questões temporais (aumentando a coerência do modelo), no modelo R^2 elas são consideradas independentemente. Embora o esquema de concorrência utilizado em R^2 permita o tratamento concorrente dos pedidos de ativação de métodos (scripts), estes pedidos serão liberados na ordem de chegada e só posteriormente, quando o método liberado estiver em execução, é que seus segmentos com restrições temporais associadas serão considerados pelo meta-objeto Scheduler. Por outro lado, a sincronização condicional não é controlada reflexivamente no modelo R^2.

Conjuntamente, estas diferenças fazem com que o modelo RTR seja mais expressivo na representação e mais efetivo no controle dos aspectos temporais das aplicações.

O RT-MOP (Real-Time Meta-Object Protocol) proposto recentemente em [Mitchell 97], tem como objetivo permitir que o comportamento temporal das aplicações possa ser controlado e modificado dinamicamente. Segundo o RT-MOP proposto, as aplicações são estruturadas em grupos e meta-grupos de escalonamento, onde cada meta-grupo possui um meta-objeto responsável pelas decisões de escalonamento; tais meta-objetos podem implementar diferentes políticas de escalonamento e oferecer níveis diferenciados de garantias. Além disso, os meta-grupos também são reflexivos e podem ser modificados dinamicamente por um meta-meta-grupo. Esta capacidade é interessante sob o ponto de vista da flexibilidade e, embora não seja suportada diretamente pelo modelo RTR, pode ser simulada através de meta-objetos escalonadores que implementem diversas políticas de escalonamento (como descrito na seção III.5.8). Outros aspectos tais como representação e controle de restrições temporais não são explicitados na referência citada.

representadas e controladas segundo a abordagem proposta. Em particular, a utilização do modelo RTR na programação das restrições de sincronização típicas de aplicações multimídia foi explorada e exemplificada detalhadamente.

Dentre as principais vantagens do modelo proposto, além de sua expressividade, destacamos sua flexibilidade, sua capacidade para gerenciamento da complexidade e sua independência de suporte e de ambiente operacional relativa ao controle de restrições temporais das tarefas de uma aplicação e ao escalonamento tempo real destas tarefas. Estas vantagens decorrem diretamente da estrutura reflexiva proposta, a qual permite que aspectos temporais sejam integrados ao modelo de objetos, sem prejuízo dos benefícios deste modelo.

Por outro lado, a principal limitação do modelo é a dificuldade para obtenção de previsibilidade em função da flexibilidade oferecida; devido a isso, a utilização do modelo na programação de sistemas tempo real *hard* é bastante restrita. Além disso, o *overhead* devido a reflexão, embora aceitável e gerenciável no caso geral, pode se constituir em uma limitação nos casos onde a taxa de utilização de recursos de processamento seja extremamente elevada.

No próximo capítulo serão comentadas várias questões genéricas relativas a implementação do modelo proposto e será apresentada a especificação de Java/RTR, uma linguagem de programação tempo real que estende a linguagem Java (Sun Microsystems Inc.) e implementa explicitamente a estrutura e o comportamento do modelo RTR.

Capítulo IV - Uma implementação do Modelo RTR : A Linguagem Java/RTR

IV.1 - Introdução

O modelo RTR caracteriza-se por ser uma abstração capaz de impor uma filosofia de programação tempo real baseada nos paradigmas de objetos e reflexão computacional. Entretanto, para que esta filosofia possa ser colocada em prática, o modelo proposto deve possuir uma implementação concreta que estabeleça a disciplina de programação necessária para obtenção do comportamento especificado em sua definição.

Neste sentido, além da implementação experimental da extensão distribuída do modelo RTR para ambientes abertos [Siqueira 96] apresentada no capítulo anterior, uma segunda experiência de implementação (a ser descrita na próxima seção) foi realizada através do mapeamento do modelo RTR sobre a linguagem de programação Java [Nishida 96]. Com base nestas experiências, definimos uma implementação completa do modelo RTR através da proposição de uma linguagem de programação denominada Java/RTR. A linguagem proposta é uma extensão da linguagem Java que incorpora explicitamente a estrutura e a dinâmica de funcionamento do modelo a nível de linguagem, permitindo a programação sistemática de aplicações tempo real segundo a filosofia de programação do modelo RTR.

Neste capítulo, apresentamos algumas considerações gerais sobre implementação do modelo RTR, relatamos uma experiência preliminar relativa ao mapeamento do modelo RTR sobre a linguagem Java e descrevemos a linguagem Java/RTR. Na descrição de Java/RTR, além da especificação propriamente dita, fazemos algumas considerações sobre Java - a linguagem base utilizada, e sobre o pré-processador a ser utilizado na tradução de programas Java/RTR para programas Java eqüivalentes. Adicionalmente, apresentamos uma proposta preliminar para análise do tempo de execução de programas Java/RTR. Encerrando o capítulo, apresentamos uma análise comparativa entre Java/RTR e outras linguagens tempo real orientadas a objetos existentes.

IV.2 - Considerações gerais sobre implementação do Modelo RTR

Nesta seção, apresentamos de forma genérica alguns aspectos básicos a serem considerados em uma implementação do modelo, os quais juntamente com as experiências realizadas podem servir de guia para futuras implementações do modelo. Os aspectos a serem abordados nesta seção incluem: formas de implementação da estrutura reflexiva do modelo; estruturação das aplicações; questões relativas a concorrência e sincronização; e implementação de classes padronizadas para representação do comportamento básico dos diferentes tipos de meta-objetos que constituem o modelo. Complementarmente são descritos os principais aspectos de uma implementação do modelo RTR realizada através da abordagem de simulação.

IV.2.1 - Abordagens para implementação da estrutura reflexiva do modelo

Em termos práticos uma implementação do modelo RTR corresponde a uma implementação de sua estrutura reflexiva e de seu comportamento temporal em uma linguagem de programação. Diversas são as abordagens através das quais isto pode ser realizado:

1 - Definição de um MOP (Meta-Object Protocol) que estabeleça o comportamento e a interação entre os componentes (objetos-base e meta-objetos) do modelo. Tal MOP pode ser definido através de uma hierarquia de classes, implementada em bibliotecas especiais, as quais podem ser usadas diretamente na programação das aplicações (via chamadas de seus métodos) ou pode ser explicitamente integrada a uma linguagem de programação através de diretivas de compilação (e de um pré-processador associado), como é o caso de Open C++ [Chiba 93a], por exemplo.

2 - Simulação do comportamento reflexivo do modelo através das estruturas disponíveis na linguagem utilizada; nesta abordagem, a conformidade com o modelo proposto depende de uma disciplina de programação e é de responsabilidade do programador da aplicação. Esta abordagem foi usada na implementação da extensão distribuída do modelo RTR para ambientes abertos [Siqueira 96] descrita no capítulo anterior, e na implementação do modelo RTR sobre a linguagem Java [Nishida 96] a ser descrita nesta seção.

3 - Extensão de uma linguagem existente, através da incorporação da sintaxe e da semântica necessárias para representação da estrutura e do comportamento do modelo. Esta abordagem subentende a definição de um novo compilador (ou extensão do compilador da linguagem base existente) ou de um pré-processador responsável pelo mapeamento da sintaxe e da semântica introduzidas para a sintaxe e semântica da linguagem base utilizada. A linguagem Java/RTR apresentada na próxima seção, segue esta abordagem.

4 - Definição de uma nova linguagem de programação onde a estrutura do modelo seja representada sintaticamente de forma explícita e seu comportamento seja garantido via semântica da linguagem.

Embora qualquer das abordagens acima especificadas (ou uma combinação delas) possa ser usada na implementação do modelo, todas apresentam vantagens e desvantagens, dependendo do ponto de vista considerado. Assim sendo, a escolha da abordagem deve ser decidida caso a caso, levando-se em consideração as finalidades e as aplicações alvo que motivaram a realização de uma implementação particular, além de fatores como disponibilidade de tempo e de recursos.

IV.2.2 - Estrutura e comportamento das aplicações

Dentre as questões a serem decididas na implementação do modelo RTR, duas influenciam diretamente a estruturação das aplicações: 1 - utilização ou não de objetos convencionais no nível-base (em conjunto com os objetos-base de tempo real); e 2 - forma pela qual a estrutura do modelo será representada no código da aplicação.

Com relação a primeira questão, uma implementação particular do modelo poderá suportar tanto a programação de aplicações estruturadas unicamente a partir de objetos-base de tempo real (OBTR), tolerando a presença de objetos convencionais na implementação de funções auxiliares e de suporte (desde que os mesmos sejam sempre ativados a partir de

OBTR), quanto aplicações nas quais coexistem OBTR e objetos convencionais. Esta última forma de estruturação apresenta vantagens (reutilização de software desenvolvido para aplicações não tempo real) e desvantagens (maior dificuldade na obtenção do comportamento temporal desejado, em função da falta de controle sobre o escalonamento e a execução dos métodos dos objetos convencionais). Entretanto, visando reduzir o impacto das desvantagens desta forma de estruturação, os seguintes procedimentos alternativos podem ser realizados:

- associação automática de um meta-objeto gerenciador aos objetos convencionais, permitindo que os mesmos tenham um tratamento similar aos OBTR sob o ponto de vista do escalonamento, porém com prioridade inferior à destinada aos OBTR. A necessidade de recompilação dos objetos convencionais para geração de código reflexivo limitará o reuso destes objetos ao caso onde o código fonte estiver disponível.

- vincular o uso de objetos convencionais aos objetos-base tempo real da aplicação; embora viável, esta alternativa exige uma disciplina de programação a qual nem sempre poderá ser garantida sistematicamente.

Com relação a segunda questão, a estrutura do modelo pode estar explícita ou implicitamente presente no código das aplicações tempo real desenvolvidas. No caso explícito, a linguagem que está sendo definida ou estendida para implementar o modelo deverá prover sintaxe e semântica apropriadas para a representação da estrutura do modelo e de suas construções temporais; desta forma a conformidade entre as aplicações desenvolvidas e o modelo implementado poderá ser verificada sistematicamente via compilação. No caso implícito, a estrutura e o comportamento do modelo devem ser simulados através de bibliotecas especiais disponíveis na linguagem a ser utilizada; desta forma a obtenção do comportamento desejado dependerá de uma disciplina de programação.

IV.2.3 - Concorrência e sincronização

Segundo o modelo RTR concorrência e sincronização devem ser tratadas reflexivamente a nível de meta-objetos gerenciadores. A seguir são apresentadas as formas para tratamento destas questões nas diferentes abordagens de implementação citadas anteriormente.

No caso de linguagens novas, a concorrência pode ser provida diretamente na definição da linguagem (via sintaxe, semântica e rotinas de suporte) ou ser integrada à linguagem através do uso de pacotes de *threads* na programação do meta-objeto gerenciador. Da mesma forma, a sincronização condicional poderá ser provida por mecanismos definidos na linguagem ou ser programada diretamente na seção de sincronização do meta-objeto gerenciador.

No caso de extensões de linguagens que não suportam concorrência, a situação é similar ao caso anterior. Já no caso de extensões de linguagens concorrentes, ou adota-se o esquema de concorrência/sincronização existente ou impõe-se o esquema definido no modelo RTR e proíbe-se a utilização do esquema existente na linguagem-base.

Nos casos de simulação e uso de bibliotecas, concorrência e sincronização podem ser tratadas ou utilizando-se pacotes de *threads* como em [Siqueira 96] ou, no caso de linguagens concorrentes, mapeando-se o esquema proposto pelo modelo RTR para os mecanismos disponíveis, como em [Nishida 96].

IV.2.4 - Utilização de meta-objetos padrão

Independentemente da abordagem utilizada, uma implementação do modelo deverá disponibilizar classes padrão representando as funcionalidades básicas dos diferentes meta-objetos (gerenciador, escalonador e relógio) presentes no modelo RTR. Tais classes poderão ou não ser distinguidas sintaticamente, dependendo da abordagem de implementação utilizada. Por exemplo, nos casos de linguagens novas ou de extensões de linguagens os diferentes componentes do modelo podem ter sintaxe própria, como ocorre no caso de Java/RTR, enquanto que nas demais abordagens estes componentes deverão ser mapeados para classes convencionais da linguagem utilizada.

As classes padrão existentes poderão ser utilizadas diretamente na instanciação dos meta-objetos, ou poderão ser especializadas (estendidas) visando melhor satisfazer as necessidades da aplicação; neste último caso, os meta-objetos deverão ser instâncias das subclasses criadas. Adicionalmente, alguns tipos de restrições temporais freqüentemente utilizados (como por exemplo *Periodic, Aperiodic e Sporadic*) podem ser predefinidos e implementados diretamente na classe MOG-padrão, não precisando ser declarados nos objetos-base da aplicação.

IV.2.5 - Uma implementação baseada na abordagem de simulação : Mapeamento do modelo RTR sobre a linguagem Java

Com o objetivo de validar o modelo RTR e avaliar a viabilidade de sua realização prática através da linguagem Java (TM Sun Microsystems Inc.), foi realizada uma implementação do modelo RTR sobre esta linguagem [Nishida 96]. Nesta implementação, a estrutura e o comportamento do modelo RTR foram programados utilizando-se exclusivamente as facilidades existentes na versão 1.02 de Java. Os principais aspectos do modelo RTR foram assim programados:

- os objetos-base e meta-objetos do modelo RTR foram mapeados para objetos Java;

- a reflexividade foi simulada através da ativação explicita do meta-objeto gerenciador;

- a criação de meta-objetos foi feita de maneira explícita;

- a associação de restrições temporais e manipuladores de exceção aos métodos dos objetos-base foi simulada na forma de descritores desses métodos;

- a concorrência do modelo RTR foi implementada usando a capacidade *multithreading* de Java;

- o controle de exclusão mútua na execução dos métodos de um objeto-base foi programado explicitamente no meta-objeto gerenciador e o controle de sincronização foi realizado com base em máquinas de estados finitos;

- o comportamento especificado pelo modelo RTR, foi obtido através de um esquema de *Threads* e prioridades, manipuladas explicitamente no nível meta;

A implementação acima especificada nos permitiu tirar as seguintes conclusões:

- a representação do modelo RTR sobre a linguagem Java é viável graças a sua capacidade *multithreading* e a existência de um escalonador a nível de linguagem;

- a representação de restrições temporais e sua associação aos métodos, mostrou-se problemática usando-se apenas as facilidades Java (versão 1.02); assim sendo, a simulação

através de descritores definidos e mantidos explicitamente pelos meta-objetos foi necessária. Contudo, a utilização do API para reflexão (disponível a partir da versão 1.1 de Java) facilita a solução dos problemas relativos a representação e controle de restrições temporais;

- a factibilidade do modelo proposto e a efetividade de seu comportamento com relação ao controle temporal das aplicações tempo real ficaram demonstradas nos resultados de vários testes realizados [Nishida 96];

- a utilização prática da abordagem de simulação não foi julgada adequada ao desenvolvimento de aplicações tempo real segundo a abordagem RTR, em função das seguintes limitações encontradas: necessidade de programação explicita de vários aspectos que deveriam ser transparentes; impossibilidade de uso sistemático do modelo RTR; impossibilidade de se evitar a interferência do uso explícito de *threads* no comportamento desejado; comprometimento de alguns objetivos do modelo RTR como gerenciamento da complexidade e incremento da confiabilidade; e restrição na separação entre questões funcionais e de controle, devido a simulação da reflexão.

Com base nos problemas e limitações encontrados, concluiu-se pela necessidade de se ter reflexão e suporte para representação de restrições temporais a nível de linguagem. No ítem subseqüente apresentamos a especificação de Java/RTR, uma linguagem que estende Java com suporte para reflexão computacional e com capacidade para representação e controle de restrições temporais com base no modelo RTR.

IV.3 - A Linguagem Java/RTR

IV.3.1 - Introdução

Nesta seção, apresentamos a especificação de uma linguagem de programação experimental, concebida com base em dois grandes objetivos:

1 - Ser uma linguagem de programação tempo real; ou seja, possuir características que satisfaçam os requisitos básicos das linguagens tempo real, conforme descrito na seção 4 do capítulo II;

2 - Viabilizar o uso do modelo RTR; ou seja, incorporar a nível de linguagem, a estrutura e a semântica de funcionamento deste modelo, conforme definido no capítulo anterior.

A linguagem proposta, denominada Java/RTR, é uma extensão da linguagem Java (Sun Microsystems Inc.) que implementa explicitamente o modelo RTR e se caracteriza por ser orientada a objetos, concorrente, reflexiva e de tempo real.

A opção pela extensão de uma linguagem existente, justifica-se pelo caráter experimental da linguagem desejada e pelo fato de estarmos interessados prioritariamente na proposição de uma linguagem capaz de implementar integral e explicitamente o modelo RTR, servindo como ferramenta básica para sua validação prática. Além disso, aspectos como menor tempo e esforço necessários para seu desenvolvimento e a possibilidade de realização de simulações ainda durante a especificação da linguagem, também foram levados em conta na escolha desta opção.

Por outro lado, as desvantagens inerentes a esta abordagem (tais como dificuldade na integração das extensões com a linguagem base e na obtenção do comportamento tempo real

desejado) foram minimizadas em função de algumas decisões de projeto, tais como o uso de reflexão computacional para a representação e o controle de restrições temporais e o interesse na obtenção de uma linguagem voltada prioritariamente para a programação de aplicações tempo real do tipo melhor esforço.

IV.3.2 - Java - a linguagem-base escolhida

Com base nos objetivos da linguagem a ser concebida, diversas alternativas (C++, Cm Distribuído e Open/C++) foram consideradas antes de escolhermos Java como sendo a linguagem-base a ser estendida.

C++ [Ellis 93], por ser uma linguagem orientada a objetos, eficiente e altamente difundida, foi considerada uma alternativa potencial; entretanto, C++ apresenta uma série de aspectos indesejáveis sob a ótica dos nossos objetivos, tais como falta de segurança (decorrente do uso de ponteiros e de seu sistema de tipos) e sobretudo a ausência de mecanismos de concorrência e de suporte para distribuição (em sua versões mais difundidas).

Open/C++ [Chiba 93a, 93b], uma extensão reflexiva de C++, foi considerada por ter as potencialidades de C++ e prover uma estrutura reflexiva. Entretanto, o esquema reflexivo de Open/C++ não permitiu expressar diretamente a estrutura reflexiva do modelo RTR, a qual teria que ser simulada e dependeria de uma disciplina de programação. A ausência de concorrência e a falta de transparência do esquema reflexivo, foram razões adicionais para que o uso de Open/C++ fosse descartado.

Cm Distribuído [Gonçalves 94], uma extensão concorrente e distribuída de C++ desenvolvida no âmbito do projeto ASAP [ASAP 94], por herdar as principais características de C++ e por introduzir concorrência e distribuição a ela, poderia ter sido a alternativa escolhida; entretanto, por se encontrar em fase de implementação, sua utilização não foi possível.

IV.3.2.1 - Visão geral de Java

Java, uma linguagem desenvolvida recentemente pela Sun Microsystems Inc, foi escolhida para ser a linguagem base de nossa linguagem tempo real em função de sua potencialidade e da perspectiva concreta de futuras extensões e ferramentas de apoio. Java é uma linguagem de programação orientada a objetos, que se caracteriza por ser uma linguagem simples, robusta, segura, concorrente, distribuída, independente de arquitetura, portável, dinâmica e de alto desempenho [Sun 95a, 95b, 95c].

A familiaridade de Java provém do fato de que sua sintaxe deriva de C e C++, enquanto sua simplicidade deve-se a ausência de muitas características confusas e pouco usadas de C++, tais como sobrecarga de operadores e herança múltipla. Além disso, Java possui um "coletor de lixo" (*garbage collector*) automático, simplificando a programação e eliminando erros potenciais relativos ao gerenciamento de memória.

A confiabilidade de Java advém de características tais como: ênfase na verificação estática, coletor de lixo automático, tratamento de exceção, ausência de ponteiros e ausência de declarações implícitas. Além disto, Java implementa diversos mecanismos de segurança que fazem dela uma linguagem segura mesmo em ambientes baseados em rede.

Programas Java são traduzidos para código de uma máquina virtual ("*bytecodes*"), o qual pode ser interpretado ou traduzido para código nativo (visando um melhor desempenho). Em função disto, programas Java podem ser portados para quaisquer sistemas nos quais o

interpretador e o sistema de execução Java tenham sido implementados, o que garante a portabilidade da linguagem e conseqüentemente sua independência de ambiente operacional.

Além das ferramentas de compilação e de execução, o ambiente de desenvolvimento Java contém um conjunto de classes (API Java) que implementam desde funções essenciais para o desenvolvimento de programas Java até funções acessórias (destinadas a apoiar o programador Java). Várias características importantes da linguagem, tais como concorrência, distribuição e reflexão computacional, são providas através destas classes.

Com relação a concorrência, Java é uma das poucas linguagens de programação orientadas a objetos que incorpora diretamente construções de concorrência à linguagem, sem requerer outras ferramentas ou sistema de suporte; concorrência em Java é provida através da classe *Thread* e da interface *Runnable,* as quais permitem a criação, o controle (independentemente da plataforma usada) e a execução de *threads* concorrentes. Adicionalmente Java fornece meios para criação de *threads-daemons,* agrupamento de *threads,* sincronização de *threads* (através do uso de monitores), controle de prioridades e escalonamento de *threads* através de um esquema de escalonamento preemptivo baseado em prioridades fixas, mantido a nível de máquina virtual.

Distribuição em Java é suportada através de classes especiais, as quais oferecem várias facilidades para a manipulação de protocolos tais como HTTP e FTP e para o acesso de objetos remotos através de URL's. Além disso, Java dispõe dos mecanismos RMI (*Remote Method Invocation*) e *Object Serialization* [Sun 96a] os quais permitem a criação de objetos cujos métodos podem ser invocados de outras máquinas virtuais (possivelmente executando em outros computadores), viabilizando o desenvolvimento de aplicações distribuídas, sem adulterar o modelo de objetos usado em Java. Encontram-se ainda em desenvolvimento uma série de ferramentas (IDL/Java [Sun 97], por exemplo) que visam a utilização de Java no desenvolvimento de aplicações distribuídas abertas, de acordo com o padrão CORBA [OMG 96a] [Curtis 97].

Java também suporta reflexão estrutural através do API *Java-Core-Reflection* [Sun 96b], o qual permite introspecção sobre classes e objetos a nível de máquina virtual Java. Este API dispõe de facilidades para acessar e modificar campos, invocar métodos de classes e objetos e acessar e modificar elementos de *arrays*. O suporte a reflexão estrutural disponível em Java, facilitará bastante a implementação do esquema reflexivo do modelo RTR.

Todas estas características fazem de Java uma boa linguagem de programação de uso geral, particularmente adequada ao desenvolvimento de *software* extensível dinamicamente em ambientes de rede. Embora Java esteja rapidamente se difundindo como um veículo para distribuição de *software* sobre "Internet", é consenso, entre fabricante e usuários, que suas características habilitam-na para um campo mais amplo de aplicações [Nilsen 95].

Entretanto, apesar de todo o seu potencial, Java não é completamente adequada para o desenvolvimento de *software* tempo-real por não permitir a representação explícita e o controle de restrições temporais e por apresentar varias características (por exemplo, coletor de lixo automático, ligação dinâmica, recursão e *loop's* ilimitados) que impedem a construção de programas com tempo de execução previsível. Estas questões (especialmente a representação e o controle dos aspectos temporais) devem ser prioritariamente consideradas numa extensão destinada a programação de aplicações tempo-real.

IV.3.3 - Especificação de Java/RTR

Java/RTR é uma extensão da linguagem Java que implementa o modelo RTR, viabilizando a filosofia de programação introduzida por este modelo. As extensões introduzidas em Java relacionam-se direta ou indiretamente com a definição de um esquema reflexivo e com a representação e o controle de aspectos temporais. Estas extensões serão traduzidas diretamente para Java, seguindo a partir de então o fluxo normal de processamento de programas Java.

Nesta seção, especificaremos a sintaxe (usando *BNF-like*) e a semântica (descrita textualmente) das extensões propostas; complementarmente a integração destas extensões a Java serão ilustradas através de exemplos. O mapeamento destas extensões para Java será realizado por um pré-processador, cuja especificação será apresentada na seção subsequente, enquanto que a sintaxe de Java/RTR é apresentada formalmente no apêndice A.

IV.3.3.1 - Estrutura reflexiva de Java/RTR

Aplicações Java/RTR são estruturadas na forma de objetos-base e meta-objetos, refletindo fielmente a estrutura do modelo RTR. Os diferentes componentes do modelo (objetos-base e meta-objetos) são representados através de diferentes tipos de classes, estruturadas segundo sintaxe e semântica próprias que estendem a sintaxe Java de declaração de classes e implementam a semântica do modelo RTR, como especificado a seguir.

Declaração de classes em Java - A declaração de classes (**class**) em Java, tem a seguinte estrutura geral :

[<*ClassModifiers*>] **class** <*Identifier*> [**extends** <*TypeName*>]
[**implements** <*TypeNameList*>]

<*ClassBody*>

onde :

- <*ClassModifiers*> especifica os modificadores da classe, os quais podem ser *final, abstract ou public*;

- <*Identifier*> é o nome da classe;

- A cláusula **extends** especifica a superclasse imediata da classe que está sendo declarada;

- A cláusula **implements** especifica a lista de interfaces implementadas pela classe que está sendo declarada;

- <*ClassBody*> consiste de uma lista de declarações de variáveis, métodos e construtores.

Declaração de classes e meta-classes em Java/RTR - Os diferentes tipos de classes e meta-classes de Java/RTR são definidos explicitamente através da extensão da declaração de classes Java, como mostrado abaixo[1].

[<*ClassModifiers*>] [<*RTRoption*>] **class** <*Identifier*> [**extends** <*TypeName*>]
[**implements** <*TypeNameList*>]

<*ClassBody*>

---

[1] Para manter uma coerência lingüistica entre as extensões introduzidas e a linguagem base utilizada, as estruturas usadas (suas palavras-chave e meta-símbolos) serão escritas em inglês.

onde:

- *<RTRoption>* identifica o tipo de classe Java/RTR que está sendo declarada, a qual poderá ser uma:

  - classe-base "Real-Time" (***RTBC*** - Real-Time Base-Class);
  - meta-classe "Manager" (**MMC** - Manager Meta-Class);
  - meta-classe "Scheduler" (**SMC** - Scheduler Meta-Class **);**
  - meta-classe "Clock" (**CMC** - Clock Meta-Class);
  - classe-base convencional, assumida por *default* quando nenhuma das opções acima for utilizada.

- A cláusula **extends** possui semântica específica para cada tipo de classe Java/RTR;

- *<ClassBody>* estende as possibilidades Java com sintaxe/semântica específicas de cada um dos tipos de classes Java/RTR.

Ao diferenciar sintaticamente os vários tipos de classes que podem ser utilizadas no desenvolvimento de seus programas, Java/RTR possibilita a verificação sistemática da conformidade entre a estrutura das aplicações e a estrutura do modelo RTR. Além disso, esta diferenciação facilita a realização de vários procedimentos semânticos relativos a análise das extensões introduzidas e ao mapeamento destas extensões para código Java correspondente.

Por outro lado, não há como garantir sistematicamente que as meta-classes utilizadas na aplicação comportem-se de acordo com a definição do modelo RTR, uma vez que estas meta-classes são parte integrante da aplicação e portanto suas implementações (e conseqüentemente a obtenção do comportamento desejado) ficam sob a responsabilidade do programador da aplicação. Em função disso e visando auxiliar (e mesmo dirigir) o programador na obtenção do comportamento desejado, Java/RTR dispõe de Interfaces e Classes padronizadas pré-definidas, as quais serão descritas nas seções subseqüentes.

IV.3.3.1.1 - Classes-base "Real-Time" (RTBC)

Estrutura geral - As classes RTBC diferenciam-se de classes convencionais Java por suportarem a declaração de tipos de restrições temporais e a associação de restrições temporais e de manipuladores de exceção temporal à declaração de seus métodos. A estrutura geral de uma classe RTBC é a seguinte:

[*<ClassModifiers>*] **RTBC class** *<Identifier>* [**extends** *<TypeName>*]

[**implements** *<TypeNameList>*]

<ClassBody>

onde:

- *< TypeName >* deve ser uma classe RTBC ou uma classe convencional Java;

- *<ClassBody >* introduz as seguintes extensões e limitações relativas às classes convencionais Java:

  - Declaração de tipos de restrições temporais;

  - Associação de restrições temporais, categorias de controle e manipuladores de exceções temporais à declaração de métodos;

  - Associação de parâmetros temporais e da cláusula *timeout* à ativação de métodos;

• Proibição da criação e manipulação explícita de *threads*.

As extensões e limitações que compõem o corpo de uma classe RTBC serão detalhadamente especificadas e exemplificadas nas seções subsequentes.

Toda aplicação Java/RTR deverá possuir pelo menos uma classe RTBC, a qual será responsável pelo *start-up* da aplicação; esta classe deverá possuir um método *main*() que será ativado pelo interpretador Java quando da execução do programa Java resultante do pré-processamento do programa Java/RTR original.

IV.3.3.1.2 - Meta-classes "Manager" (MMC)

Estrutura geral - As meta-classes MMC de Java/RTR são as meta-classes a partir das quais os meta-objetos do tipo "Manager" serão instanciados, e são definidas da seguinte forma:

[<*ClassModifiers*>] **MMC class** < *Identifier* > [**extends** <*TypeName*>]
[**implements** <*TypeNameList*>]
{ [< *ManagementSection* >]
[< *SynchronizationSection* >]
[< *ExceptionsSection* >]
[< *TimingConstraintsSection* >] }

Onde:

- <*Identifier* > deve ser um identificador composto pelo nome de uma classe RTBC precedido pelo prefixo "Meta_", permitindo a ligação automática entre a meta-classe que está sendo declarada e sua classe RTBC correspondente;

- <*Typename* > deve ser uma meta-classe "Manager" (MMC);

- As diversas seções especificadas constituem o corpo de uma meta-classe MMC e são compostas por declarações de variáveis e métodos necessários para a realização das funções específicas de cada seção, de acordo com a especificação do modelo RTR. Estas declarações devem ser especificadas de acordo com a sintaxe e a semântica Java, com exceção das restrições de sincronização relativas aos métodos do objeto-base correspondente (*SynchronizationSection*>) que podem ser programadas explicitamente usando recursos Java ou especificadas através de uma *path-expression*; a sintaxe e semântica destas *path-expression* são próprias de Java/RTR e serão detalhadas mais adiante na seção de sincronização.

Programação de meta-classes MMC - Embora Java/RTR possua uma estrutura específica para a definição de MMC (identificada através da opção **MMC**), a programação das meta-classes MMC (sua composição interna e seu comportamento) é de responsabilidade do programador; entretanto, visando facilitar e sistematizar o desenvolvimento destas meta-classes, Java/RTR dispõe de uma interface e de uma meta-classe padrão pré-definidas, as quais são descritas a seguir.

Para que uma meta-classe MMC esteja em conformidade com o modelo RTR, ou seja, possibilite que os meta-objetos "Manager" instanciados a partir dela comportem-se da forma desejada, ela deve conter um conjunto de métodos que compõem um protocolo mínimo de gerenciamento especificado através da interface "Protocol-MMC" (figura 4.1). Isto equivale a dizer que toda meta-classe MMC de uma aplicação deve implementar a interface "Protocol-

MMC" diretamente, ou estender uma outra meta-classe MMC que implemente esta interface. A verificação desta condição será realizada sistematicamente pelo pré-processador de Java/RTR.

```
Interface Protocol-MMC;
{       // métodos usados na recepção e encaminhamento de pedidos de ativação
        public void ReceiveRequests (...);
        protected void HandleRequestWithoutTemporalConstraints (...);
        // métodos usados no controle de concorrência
        protected void ReleaseActivationRequest (...);
        protected void EndOfExecution (...);
        // métodos usados no controle de sincronização
        protected boolean VerifySynchronization (...);
        protected void UpdateSynchronizationState (...)
}
```

Figura 4.1 - Interface Protocol-MMC

Embora a existência da interface acima descrita possibilite que as meta-classes MMC da aplicação comportem-se da maneira prevista no modelo RTR, tal comportamento não é garantido e dependerá da implementação particular de cada método que compõe a referida interface. No sentido de garantir o comportamento desejado, o ambiente Java/RTR disponibilizará uma meta-classe MMC padrão, denominada "Standard-MMC", que além de implementar a interface "Protocol-MMC" também conterá a implementação de um conjunto de restrições temporais predefinidas.

Assim sendo, no caso geral, a programação de uma meta-classe MMC correspondente a uma classe RTBC da aplicação consistirá em:

- estender a meta-classe padrão "Standard-MMC" e, se for o caso redefinir alguns de seus métodos; alternativamente, dependendo das especificidades da aplicação, poder-se-á optar por uma nova implementação da interface "Protocol-MMC";

- especificar as restrições de sincronização entre os métodos da classe RTBC correspondente através de uma *Path-Expression* ou de programação explícita;

- implementar os novos tipos de restrições temporais declarados na classe RTBC correspondente;

- implementar os manipuladores de exceção associados aos métodos (com restrição temporal) da classe RTBC correspondente;

- implementar, se for o caso, os procedimentos de controle relativos as diferentes categorias de controle usadas na declaração dos métodos da classe RTBC em questão.

IV.3.3.1.3 - Meta-Classes "Scheduler" (SMC)

De acordo com o modelo RTR, os meta-objetos "Scheduler" de Java/RTR tem a função de implementar uma política de escalonamento e liberar ativações de métodos do objeto base de acordo com esta política, objetivando influenciar no escalonamento final das

atividades do sistema. Em cada nodo do sistema computacional onde a aplicação estiver sendo executada deverá haver um meta-objeto "Scheduler", o qual deverá ser uma instância de uma meta-classe SMC definida como sendo:

[<*ClassModifiers*>] **SMC class** < *Identifier* > [**extends** <TypeName>]
[**implements** <TypeNameList>]
{ < *ClassBody* > }

onde:

- < *super* > deve ser uma meta-classe "Scheduler" (SMC);

- <*ClassBody*> é o corpo de uma meta-classe "Scheduler", o qual a exemplo de uma meta-classe "Manager"(MMC), também deve satisfazer um protocolo mínimo capaz de implementar as funções de escalonamento previstas na definição do modelo RTR. Este protocolo é especificado através de uma interface Java, denominada "Protocol-SMC" (figura 4.2), a qual deverá ser implementada (direta ou indiretamente) pela SMC usada em cada aplicação. Na implementação destas meta-classes podem ser usadas todas as facilidades Java.

```
Interface Protocol-SMC
{
        // método responsável pelo recebimento de pedidos de escalonamento e
        // pela  implementação de uma política de escalonamento
        public void Schedule (...);
        // libera pedido de ativação de acordo com a política implementada
        public void ReleaseNextRequest (...);
        // método ativado pelo meta-objeto "Clock" quando o deadline de
        // um método requisitado é violado
        public void RemoveFromSchedulingQueue (...)
}
```

Figura 4.2 - Interface Protocol-SMC

Como diferentes políticas são implementadas através de diferentes meta-classes SMC, o ambiente Java/RTR disponibilizará diferentes SMC predefinidas (implementando as políticas de escalonamento mais comumente utilizadas). Novas políticas poderão vir a ser utilizadas pelo programador, através de novas implementações da interface Protocol-SMC ou da redefinição das meta-classes SMC pré-definidas.

IV.3.3.1.4 - Meta-Classes "Clock" (CMC)

Os meta-objetos "Clock" são meta-objetos usados para programação de ativações de métodos em um tempo futuro e para a execução destas ativações quando o tempo programado chegar. Em cada nodo do sistema computacional onde uma aplicação Java/RTR estiver sendo executada, deverá haver um meta-objeto "Clock", o qual deverá ser uma instância de uma meta-classe "Clock" (CMC) assim definida :

[<*ClassModifiers*>] **CMC class** < *Identifier* > [**extends** <TypeName>]
[**implements** <TypeNameList>]

{ < *ClassBody* > }

onde:

- < *super* > deve ser uma meta-classe "Clock" (CMC);

- <*ClassBody*> é o corpo de uma meta-classe "Clock", o qual da mesma forma que nos casos anteriores, deve implementar um protocolo mínimo para que o comportamento previsto no modelo RTR seja garantido. Este protocolo é especificado através de uma interface Java, denominada "Protocol-CMC" (figura 4.3), a qual é implementada através de uma CMC padrão, denominada "Standard-CMC" disponível no ambiente Java/RTR. Assim sendo, a nível de aplicação, o usuário poderá criar o meta-objeto "Clock" diretamente da meta-classe CMC padrão, estender esta meta-classe ou optar por uma nova implementação de "Protocol-CMC".

```
Interface Protocol-CMC
{       // agenda pedidos de ativações para um tempo futuro
        public void ProgramFutureActivation (...);
        // Efetua ativações de acordo com a programação atual
        private void Activate (...);
        // Cancela programação de ativações
        public void CancelProgrammedActivation (...)
}
```

Figura 4.3 - Interface Protocol-CMC

IV.3.3.1.5 - Classes convencionais

Em Java/RTR as classes convencionais de Java podem ser usadas em duas situações distintas:

1 - Para implementação de funções auxiliares que não se constituem em tarefas (tempo real ou não) da aplicação;

2 - Para implementação de tarefas não tempo real.

No primeiro caso, as classes convencionais tem natureza complementar e devem ser usadas no contexto de outras tarefas da aplicação (como parte destas tarefas); ou seja, seus métodos devem ser invocados a partir de métodos de classes RTBC, os quais deverão computar o tempo de execução dos métodos auxiliares invocados, como parte de seu próprio tempo de execução. Assim sendo, as funções implementadas por estas classes, por não se configurarem como tarefas da aplicação, não são consideradas no escalonamento realizado pelo meta-objeto "Scheduler" da aplicação, executando sempre no contexto de outras tarefas previamente escalonadas pelo referido meta-objeto.

No segundo caso, as classes convencionais utilizadas representam classes não tempo real da aplicação e como tal devem ser consideradas pelo escalonador da aplicação; isto é, seus métodos só serão escalonados pelo meta-objeto scheduler quando nenhum outro método

com restrição temporal associada tiver sido solicitado. Entretanto, para que este comportamento seja viável, tais classes devem ser declaradas como sendo do tipo RTBC. Estas classes são consideradas RTBC especiais, pois estendem classes convencionais de Java e não precisam, necessariamente, ter uma meta-classe MMC especificamente definida para elas. Neste caso, uma meta-classe MMC especial pré-definida ("SpecialMMC"), implementando apenas os métodos usados na recepção e encaminhamento de pedidos de ativação, será implicitamente associada a estas classes pelo pré-processador Java/RTR.

Por outro lado, classes RTBC especiais, também podem ser usadas para estender classes convencionais com o propósito de impor um comportamento temporal aos seus métodos, viabilizando assim a transformação (e portanto a reutilização) de classes convencionais em classes RTBC com esforço de programação adicional mínimo; neste caso, entretanto, há necessidade de definição de uma meta-classe "Manager" correspondente, implementando o comportamento temporal desejado. Na figura 4.4 é apresentado um exemplo da transformação de uma classe convencional em uma classe RTBC especial, através da redefinição ("overriden") dos métodos que devem apresentar comportamento temporal.

```
import Java.awt.*;
public class Animation extends Frame
{       ...
        public void Anima ( ... )
        { ... };
        // outros métodos
        ...
}
public RTBC class RTAnimation extends Animation
{       ...
        public void Anima ( ... ), Periodic ( Period, EndTime, MET ),
                                    AnimaException ()
        {
                super.anima ( ... );
        }
        // redefinição de outros métodos, se necessário
        ...
}
```

Figura 4.4 - Transformação de uma classe convencional em uma classe RTBC

Os motivos principais pelos quais Java/RTR suporta classes convencionais (diretamente ou na forma de RTBC especiais) são:

- evitar o *overhead* do esquema reflexivo na execução de funções auxiliares, que supostamente não usam o potencial da reflexão;

- permitir a utilização das classes que compõem o ambiente de desenvolvimento Java (Java API) diretamente e sem o *overhead* da reflexão;

- permitir a definição e a reutilização de classes convencionais que embora não exibam comportamento tempo real podem representar tarefas de aplicações tempo real.

Desta forma, Java/RTR atende uma necessidade básica de muitos sistemas tempo real que naturalmente se constituem de partes tempo real e partes não tempo real.

IV.3.3.2 - Criação de objetos-base e meta-objetos

Em Java/RTR, em função dos diferentes tipos de classes existentes, diferentes tipos de objetos devem ser criados, como descrito a seguir:

- **objetos-base "Real-Time"** são criados explicitamente através da operação "new" aplicada sobre uma classe RTBC; semanticamente a criação de um objeto deste tipo implica na criação de um meta-objeto "Manager" correspondente, como descrito no parágrafo abaixo.

- **meta-objetos "Manager"** são criados automaticamente (de forma transparente ao usuário) a partir de uma meta-classe "Manager", sempre que um objeto-base "Real-Time" for criado. Estes meta-objetos são criados implicitamente pelo pré-processador Java/RTR a partir da meta-classe MMC correspondente a classe RTBC em questão.

Por exemplo, dada a classe

```
RTBC class ClasseExemplo { ... };
```

a declaração

```
ClasseExemplo ObjetoExemplo = new ClasseExemplo();
```

resultará na criação do objeto-base "Real-Time" "ObjetoExemplo" e também na criação do meta-objeto "Manager" "Meta_ObjetoExemplo", o qual será uma instância da meta-classe "Manager" "Meta_ClasseExemplo" fornecida pelo usuário. A não existência de "Meta_ClasseExemplo" na mesma unidade de compilação onde ClasseExemplo foi declarada, implicará em um erro de compilação ou na utilização da meta-classe "SpecialMMC", dependendo se ClasseExemplo é uma classe RTBC comum ou uma classe RTBC especial.

- **meta-objeto "Scheduler" e meta-objeto "Clock"** são meta-objetos instanciados, respectivamente, a partir das meta-classes "Scheduler" e "Clock". Esta instanciação será feita implicitamente (de forma transparente ao usuário) pelo pré-processador Java/RTR. Para tanto, o programador deverá informar o nome das meta-classes a partir das quais os meta-objetos "Scheduler" e "Clock" usados na aplicação deverão ser instanciados; esta informação deverá ser fornecida como uma opção de compilação, e sua omissão implicará no uso das meta-classes padrão "StandardSMC" e "StandardCMC".

- **Objetos-Base convencionais** são criados através da operação "new" aplicada sobre uma classe convencional de Java.

IV.3.3.3 - Facilidades temporais

Fiel ao propósito de se constituir em uma implementação explícita do modelo RTR, Java/RTR provê todas as facilidades temporais que caracterizam o modelo, mantendo inclusive as mesmas estruturas propostas para representação e uso de restrições temporais. Assim sendo, nesta subseção será apresentada e detalhada a sintaxe concreta das estruturas temporais do modelo, adaptadas a sintaxe de Java; adicionalmente, a semântica correspondente a tais estruturas será descrita informalmente.

IV.3.3.3.1 - Declaração de novos tipos de restrições temporais

Em Java/RTR, novos tipos de restrições temporais são introduzidas através da declaração **RT-Type** inseridas no corpo de classes RTBC, cuja sintaxe e semântica são descritas a seguir.

RT-Type *<RTIdentifier>* = [*<RTType>* ,] (*<RTAtributesList>*)

onde:

- *<RTIdentifier>* identifica a restrição temporal que está sendo declarada; deve ser um identificador válido em Java, não ter sido previamente declarado e deverá ser o nome de um método da meta-classe MMC correspondente a classe RTBC na qual esta declaração estiver inserida.

- *<RTType>* especifica se a restrição temporal é *time-trigger* ("TT") ou *event-trigger* ("ET"), sendo que "ET" é assumido como *default*. Este atributo visa informar ao pré-processador da linguagem a forma como os métodos, aos quais as restrições temporais forem associadas, serão ativados: por relógio ("TT") ou por mensagens convencionais ("ET"). Para cada método ao qual uma restrição do tipo "TT" estiver associada, será gerado implicitamente pelo pré-processador da linguagem uma chamada a este método, cada vez que for criado um objeto da classe RTBC em questão.

- *<RTAtributesList>* especifica a lista de atributos da restrição que está sendo declarada, os quais deverão ser valorados quando da associação da restrição a um método ou quando da ativação deste método; estes atributos serão declarados como se fossem parâmetros formais, cujo tipo deverá ser um dos tipos simples de Java (*byte, int, short, boolean, float, char, double, long* ou *string*).

Por exemplo, as seguintes declarações introduzem novos tipos de restrição temporal a um programa Java/RTR:

a) **RT-Type** Start-at = (Real StartTime, MET);

b) **RT-Type** Start-at-TT = "TT", (Real StartTime, MET);

c) **RT-Type** Periodic-ET = (Real Period, EndTime, MET);

d) **RT-Type** Time-Polymorphic = (Real Deadline, String IdMet1, IdMet2, IdMet3);

Restrições temporais predefinidas - Além de permitir a definição de novas restrições temporais, Java/RTR disponibiliza ao usuário um conjunto de restrições temporais predefinidas, as quais podem ser associadas diretamente à declaração dos métodos de uma classe RTBC sem terem sido introduzidas pela declaração RT-Type. Este conjunto é composto pelas restrições temporais clássicas *Periodic, Aperiodic* e *Sporadic*, as quais são implementadas na meta-classe "StandardMMC" a partir da seguinte especificação:

1 - Periodic = "TT", (Real Period, EndTime, MET)

2 - Aperiodic = (Real Deadline, MET)

3 - Sporadic = (Real Deadline, TimeInterActivation, MET)

IV.3.3.3.2 - Declaração de métodos com restrições temporais

Java/RTR estende a declaração de métodos convencionais de Java, através da associação de restrições temporais, manipuladores de exceções temporais e categorias de controle aos métodos das classes RTBC, conforme especificado a seguir.

Declaração de métodos em Java - A declaração de métodos em Java tem a seguinte estrutura geral:

[<*MethodModifiers*>] <*ResultType*> <*MethodIdentifier*> ([<*ParameterList*>])

[**throws** <*TypeNameList*>]

<*MethodBody*>

onde:

- <*MethodModifiers*> refere-se aos modificadores de acesso do método (*public*, *protected* e *private)* e a outros modificadores de método tais como *static, abstract* e *final*, de acordo com o padrão Java;

- <*ResultType*> especifica o tipo do valor que o método retornará ou usa "*void*" para indicar que nenhum valor será retornado;

- <*MethodIdentifier*> identifica o método que está sendo declarado;

- <*ParameterList*> especifica a lista de parâmetros formais do método que está sendo declarado;

- A cláusula "***throws***", especifica os tipos de exceções funcionais que podem ocorrer na execução do método;

- <*MethodBody*> contém um bloco de declarações e comandos que implementam a funcionalidade do método ou, no caso de ser um método abstrato, é representado por ";".

Declaração de métodos em Java/RTR - Em Java/RTR a declaração de métodos em uma classe RTBC possui a seguinte estrutura geral:

[<*MethodModifiers*>] <*ResultType*> <*MethodIdentifier*> ([<*ParameterList*>])

[**throws** <*TypeNameList*>]

[,<*TimingConstraint* >]

[,<*TimingExceptionHandler*>]

[, **category** = <*CategoryIdentifier*>]

<*MethodBody*>

onde:

- <*TimingConstraint*> identifica a restrição temporal (nova ou predefinida) que está sendo associada ao método e, opcionalmente, pode estabelecer valores *default* para os diversos atributos que compõem a restrição em questão. A descrição sintática e semântica da associação de restrições temporais aos métodos dos objetos será apresentada mais adiante nesta seção;

- <*ExceptionHandler*> identifica o manipulador de exceções a ser implementado na meta-classe "Manager" (MMC) correspondente a classe RTBC na qual o método em questão está sendo declarado;

- a cláusula **category** associa o método que está sendo declarado a categoria de controle identificada por <*CategoryIdentifier*>. Uma categoria de controle especifica procedimentos de controle específicos a serem realizados quando métodos pertencentes a esta

categoria forem ativados (por exemplo controle estatístico, controle de qualidade de serviço e depuração); estes controles deverão ser implementados na MMC correspondente.

Associação de restrições temporais aos métodos de uma classe RTBC - A sintaxe usada para associação de uma restrição temporal (*<TimingConstraint>*) à declaração de um método é definida como sendo:

<RTIdentifier> (*<AtributesList>*)

onde:

<RTIdentifier> - deve ser um identificador de uma restrição temporal pré-definida *(Periodic, Aperiodic* ou *Sporadic)* ou introduzida através da declaração **RT-Type;**

<AtributesList> - identifica os atributos da restrição temporal que serão utilizados no contexto do método que está sendo declarado. Estes atributos poderão ou não ter os mesmos nomes usados na declaração da restrição temporal; entretanto, deve haver correspondência em número e tipo entre estes e aqueles especificados na declaração da restrição temporal em questão. Opcionalmente, poderão ser atribuídos valores *default* a estes atributos através da sintaxe

<Atribute> = *<DefaultValue>*

onde *<DefaultValue>* deve ser uma constante de tipo compatível com <Atribute>.

Exemplos - A seguir são apresentados e comentados alguns exemplos de declaração de métodos em Java/RTR. As restrições temporais usadas nestes exemplos são as restrições pré-definidas ou aquelas declaradas na seção IV.3.3.3.1.

a) **void** DisplaySound (...), *Periodic* (Period=40, EndTime=400000, MET=10),

DisplaySoundError(),

category = Statistics

{ ... }

Esta declaração associa a restrição temporal predefinida *Periodic* ao método *DisplaySound();* esta restrição estabelece que o método declarado deverá ser ativado a cada *Period* ms, e como ela é do tipo "TT", a primeira ativação ocorrerá automaticamente cada vez que for criado um objeto a partir da classe na qual o método declarado se encontra. Neste exemplo, os valores (em milissegundos) dos atributos *Period, EndTime* e *MET* são especificados por *default.* Adicionalmente, o manipulador de exceções *DisplaySoundError()* é associado ao método que está sendo declarado. Além disso, segundo esta declaração o método *DisplaySound()* pertence a categoria "Statistics", o que implica que a cada ativação do método em questão determinados procedimentos estatísticos, implementados na meta-classe "Manager" correspondente, deverão ser executados. Complementarmente, esta declaração de método não declara nenhuma exceção funcional e assume os modificadores *default* de Java.

b) **void** X(...), *Start-at* (StartTime, MET=10.), ExcX() { ... }

A restrição *Start-at* estabelece o tempo mais cedo no qual um método pode ser executado após ser ativado explicitamente (pois foi declarada como sendo do tipo "ET"). Neste exemplo, *X()* só poderá ser executado *StartTime* ms após sua ativação, sendo que o valor do atributo *StartTime* deverá ser especificado em cada ativação de *X();* adicionalmente

esta declaração especifica que o tempo máximo de execução (MET) de *X()* é de 10 ms (valor *default*) e que o manipulador de exceções *ExcX()* é associado ao método declarado.

c) **void** Y(...), *Start-at-TT* (StartTime=3000, MET=15.), ExcY() { ... }

A restrição *Start-at-TT* estabelece o tempo mais cedo no qual um método pode ser executado após uma instanciação de um objeto da classe na qual o método se encontra (restrição do tipo "TT"). Segundo esta declaração o método *Y()* de um determinado objeto-base será ativado 3000 ms após a instanciação desse objeto-base; adicionalmente é estabelecido que o tempo máximo de execução de *Y()* é de 15 ms e que *ExcY()* é o seu manipulador de exceções associado.

d) **void** SoundTransfer(...), *Periodic-ET* (Period, EndTime, MET=20.),
ExcSoundTransfer() { ... }

A restrição *Periodic-ET* estabelece que, após ser ativado explicitamente uma primeira vez (restrição do tipo "ET"), um método deverá ser ativado automaticamente a cada período *Period*, enquanto o tempo *EndTime* não for atingido. Neste exemplo, o tempo máximo de execução do método *SoundTransfer()* é de 20 ms e os valores dos atributos *Period* e *EndTime* devem ser especificados por ocasião da ativação explícita de *SoundTransfer()*; complementarmente, o manipulador de exceções *ExcSoundTransfer()* é associado ao método que está sendo declarado.

e) **void** DisplayImage(...), *Time-Polymorphic* (Deadline, IdMet1="DI-Quality1",
IdMet2="DI-Quality2", IdMet3="DI-Quality3"),
ExcDisplayImage() { ... }

A restrição *Time-Polymorphic* associada ao método *DisplayImage()*, estabelece que uma ativação deste método implicará automaticamente na ativação de uma das versões disponíveis (DI-Quality1, DI-Quality2 ou DI-Quality3); a escolha de uma destas versões, dependerá da disponibilidade de tempo no momento da ativação com relação ao *deadline* especificado. Neste exemplo o valor do atributo *deadline* deverá ser especificado quando da ativação do método *DisplayImage()*; adicionalmente, considera-se que o *MET* das diferentes versões que implementam *DisplayImage()* é provido na declaração destas versões; complementarmente o manipulador de exceções *ExcDisplayImage()* (a ser ativado quando nenhuma das versões disponíveis puder ser executada) é associado ao método que está sendo declarado.

IV.3.3.3.3 - Ativação de métodos com restrições temporais

Java/RTR suporta a ativação de métodos com restrição temporal através da especificação de valores a serem atribuídos aos atributos das restrições temporais na forma de um segundo conjunto de parâmetros atuais. Tais valores devem corresponder em número e tipo com os atributos especificados na declaração da restrição temporal em questão; além disso, deve ser observada a relação posicional dos mesmos (considerando-se a presença de atributos com valor *default*). Sintaticamente, a ativação de um método com restrição temporal associada terá a seguinte forma geral:

<ObjectIdentifier>.<MethodIdentifier>(<ParameterList>),(<TimingParameterList>)

Os métodos "b", "d" e "e" do exemplo anterior poderiam ser ativados, respectivamente, através dos seguintes comandos de ativação:

<id-objeto> . X(...), (1000);

<id-objeto> . SoundTransfer(...), (90, 4500);

<id-objeto> . DisplayImage(...), (100);

A não correspondência em número e/ou tipo entre os atributos temporais de uma restrição temporal e os valores atuais destes atributos, causará um erro de compilação. Entretanto, deve ser observado que os atributos de uma restrição temporal podem possuir valores *default* (atribuídos na associação da restrição ao método), sendo opcional o fornecimento de valores atuais; contudo, neste caso, a posição dos atributos deve ser considerada.

IV.3.3.3.4 - Cláusula *timeout*

Introduzida na extensão distribuída do modelo RTR apresentada no capítulo anterior, a cláusula *timeout* tem como objetivo limitar o tempo de espera pelo resultado da execução de um método remoto ativado sincronamente. Embora distribuição não esteja sendo considerada nesta primeira versão de Java/RTR, a presença da cláusula *timeout* deve-se a nossa intenção de futuramente utilizar Java/RTR na programação de aplicações tempo real distribuídas. A sintaxe desta cláusula é apresentada na figura 4.5 e sua semântica é a mesma definida informalmente na seção III.7.

```
<ObjectIdentifier>.<MethodIdentifier> ([<ParameterList>]) [,(<TimingParameterList>)] ,
        Timeout ( < TimeoutValue > ),
                {
                        case timeout : < ExceptionHandler >
                        [case reject  : < ExceptionHandler >]
                        [case abort   : < ExceptionHandler >]
                }
```

Figura 4.5 - Estrutura da cláusula *timeout* em Java/RTR

IV.3.3.4 - Interação entre objetos

A interação entre os diferentes tipos de objetos e meta-objetos que compõem uma aplicação Java/RTR, está sujeita às seguintes restrições (as quais são verificadas implicitamente pelo pré-processador Java/RTR):

1 - objetos-base "Real-Time" só podem ativar métodos de objetos convencionais e de objetos-base "Real-Time";

2 - meta-objetos "Manager" podem ativar métodos de quaisquer tipos de objetos e meta-objetos; entretanto, com relação a objetos-base "Real-Time", um meta-objeto "Manager" só pode acessar o seu objeto-base correspondente;

3 - meta-objetos "Scheduler" e meta-objetos "Clock" não podem ativar métodos de objetos-base "Real-Time";

4 - Objetos convencionais devem acessar apenas métodos de objetos convencionais.

Com exceção das interações entre objetos-base "Real-Time" (que como será visto mais adiante apresentam sintaxe e semântica próprias de Java/RTR), todas as demais interações permitidas seguem a sintaxe e a semântica de Java convencional.

A interação entre objetos-base "Real-Time" dá-se de forma **reflexiva** através de mensagens (ativações de métodos) síncronas e assíncronas. Mensagens síncronas possuem a mesma sintaxe de ativação de métodos de Java, enquanto mensagens assíncronas são diferenciadas sintaticamente pela presença do símbolo "@" precedendo um comando de ativação de métodos

Semanticamente, a ativação (síncrona ou assíncrona) de um método de um objeto-base "Real-Time" é traduzida pelo pré-processador Java/RTR para uma ativação do método *ReceiveRequest()* do meta-objeto "manager" correspondente ao objeto-base que está sendo acessado. Além disso, caso o método ativado possua uma restrição temporal associada, será verificado se o conjunto de valores para os atributos temporais de tal restrição estão sendo fornecidos corretamente.

No caso específico de chamadas assíncronas, cada chamada implicará na criação de uma *thread* de controle, na qual o método solicitado será executado paralelamente ao método chamador. A existência de chamadas assíncronas em Java/RTR torna a criação de *threads* transparente ao programador, facilitando a utilização das mesmas. O uso de chamadas assíncronas em Java/RTR visa incrementar a concorrência e facilitar a implementação e o uso de métodos com restrição temporais.

IV.3.3.5 - Concorrência e sincronização

Java é uma linguagem *multithread* com *threads* controladas através de monitores e variáveis condição. Por outro lado, o modelo RTR estabelece que os objetos-base devem ser *mono-threads* enquanto que os meta-objetos devem ser *multithreads.* Além disso, segundo o modelo RTR a concorrência em objetos-base "Real-Time" deve ser controlada pelos respectivos meta-objetos "manager", os quais ficam responsáveis tanto pela exclusão mútua na execução dos métodos do objetos-base quanto pelo controle da sincronização condicional entre estes métodos.

Uma outra diferença existente entre os dois esquemas de concorrência, é a forma como *threads* podem ser criadas e manipuladas: em Java, *threads* podem ser criadas arbitrariamente e manipuladas livremente pelo programador; no modelo RTR, para que o comportamento desejado possa ser obtido, deve haver uma disciplina na criação e na manipulação de *threads* a nível base, uma vez que o meta-objeto "scheduler" da aplicação deve ser conhecedor de todas as *threads* em uso para poder controlá-las de acordo com as restrições temporais especificadas e a política de escalonamento utilizada..

Uma terceira diferença é a forma como sincronização condicional é obtida: em Java, um método sincronizado pode ser suspenso no meio de sua execução, aguardando por uma determinada condição, enquanto que no modelo RTR todo método liberado pelo meta-objeto "Scheduler" deve concluir sua execução (uma vez iniciada) de forma incondicional. Observe-se que isto não significa que o método não possa ser preemptado por outros motivos, tais como solicitação de um serviço remoto (no caso de implementações distribuídas) ou chegada de um pedido (para outro objeto-base!) mais prioritário, desde que o meta-objeto "Scheduler" tenha conhecimento e possa gerenciar a preempção ocorrida (o que não é o caso de Java, onde os monitores são controlados pela máquina virtual).

Assim sendo, na definição de Java/RTR optamos por uma abordagem mista, permitindo simultaneamente o uso das facilidades Java sem restrições onde for possível e o uso do esquema proposto no modelo RTR onde for necessário. Desta forma, concorrência e sincronização em Java/RTR deve ser disciplinado de acordo com as seguintes regras:

1 - **Programação de meta-objetos** - Na programação de meta-objetos as facilidades e os mecanismos de Java podem ser usados integralmente. Por exemplo, na programação do meta-objeto "Clock" é imprescindível a existência de uma *thread* especial destinada a controlar ativações *time-trigger*; outro exemplo é a necessidade de sincronização entre vários métodos que implementam as funcionalidades dos meta-objetos "Manager" e "Scheduler", uma vez que estes meta-objetos são *multi-thead.*

2 - **Programação de objetos-base "Real-Time"** - Visando prover o comportamento especificado pelo modelo RTR, objetos-base "Real-Time" não poderão criar e manipular threads explicitamente e nem utilizar os monitores e variáveis condição de Java. Estas restrições implicam na proibição do uso das seguintes facilidades Java: classe *thread*, interface *Runable* e cláusula **synchronized** (e conseqüentemente os métodos *wait()* e *notify()*). Entretanto, embora o uso direto destas facilidades possa ser verificado e proibido sistematicamente pelo pré-processador Java/RTR, elas podem ser usadas indiretamente (propositalmente ou não), e neste caso torna-se necessário a adoção de uma disciplina de programação.

Em função das restrições acima especificadas, a programação de concorrência e sincronização nos objetos-base "Tempo-Real" de Java/RTR dar-se-á da seguinte forma:

- A criação de novas *threads* de controle ocorrerá implicitamente como resultado de chamadas assíncronas. Esta restrição não reduz o potencial de Java com relação a concorrência, uma vez que conceitualmente a criação de uma nova *thread* em Java corresponde a ativação assíncrona do método que representa o comportamento desta *thread* (denominado método *run*()); a diferença, entretanto, esta no controle da *thread*, que desta forma não poderá ser feito pelo programador de nível base, o que é coerente com os propósitos da reflexão computacional e justificável na programação de aplicações tempo real.

- O controle de exclusão mútua entre os métodos dos objetos-base "Real-Time" será feito explicitamente pelo meta-objeto "Manager" correspondente, e não através do mecanismo de monitores disponíveis em Java; isto se faz necessário uma vez que o meta-objeto "Scheduler" da aplicação necessita ser notificado dos bloqueios por exclusão mútua ocorridos, para que um novo método (de outro objeto-base) possa ser liberado para execução;

- A especificação e o controle de sincronização condicional deverá ser programada explicitamente na seção de sincronização do meta-objeto "Manager" ou através do mecanismo de *path-expressions.* Nesta primeira versão de Java/RTR introduzimos o mecanismo de *path-expression* em função de sua adequação a filosofia reflexiva do modelo. Segundo este mecanismo as restrições de sincronização são naturalmente separadas do código da aplicação, podendo ser completamente especificadas e verificadas a nível de meta-objeto (mais especificamente na seção de sincronização do meta-objeto "Manager").

Usando este mecanismo, as restrições de sincronização serão especificadas através da declaração de uma *path-expression* (cuja sintaxe é apresentada no apêndice "A"), onde os operandos são os identificadores dos métodos do objeto-base correspondente. Para fins de comparação são apresentadas nas figura 4.6 e 4.7, respectivamente, soluções em Java e

Java/RTR (usando *path-expression*) para a programação de um *buffer* de 1 posição a ser compartilhado entre produtores e consumidores.

```
class Buffer
{       private int contents;
        private boolean avaliable = false;
        public synchronized int get ()
        {       while (avaliable == false)
                        wait ();
                avaliable = false;
                notify ();
                return contents;
        }
        public synchronized void put (int value)
        {while (avaliable == true)
                        wait ();
                contents = value;
                avaliable = true;
                notify ();
        )
}
```

Figura 4.6 - Implementação de um buffer usando uma classe convencional Java

```
RTBC class Buffer
(       private int contents;
        public int get ();
        {
            return contents;
        }
        public void put (int value)
        {
            contents = value;
        }
}
MMC class Meta_Buffer extends Standard-MMC
{       ...
        // seção de sincronização
        path put ; get end;
        protected boolean VerifySynchronization (...)
        { ... };
        protected void UpdateSynchronizationState (...)
        { ... };
         ...
}
```

Figura 4.7 - Implementação de um Buffer usando classes RTBC e MMC de Java/RTR

3 - **Programação de objetos convencionais** - Da mesma forma que os objetos-base "Real-Time", os objetos-base convencionais utilizados em aplicações Java/RTR, não poderão usar as facilidades Java relativas a concorrência e sincronização. Assim sendo, os objetos-base que apresentam aspectos de concorrência e/ou sincronização, deverão ser programados como sendo objetos-base "Tempo-Real", utilizando as facilidades Java/RTR descritas no item anterior.

Embora esta solução seja aceitável quando os objetos convencionais necessários são próprios da aplicação, ela impede a reutilização de classes desenvolvidas fora do contexto Java/RTR que utilizem as facilidades Java relativas a concorrência e sincronização. Além disso como estas classes não serão analisadas pelo pré-processador de Java/RTR, a presença destas facilidades não poderá ser verificada sistematicamente, exigindo uma disciplina de programação.

IV.3.3.6 - Distribuição

Embora distribuição seja um dos objetivos de projeto de Java/RTR, tendo inclusive influenciado na escolha da linguagem base, esta questão não chegou a ser profundamente explorada no trabalho desenvolvido, devendo ser considerada em uma versão futura de Java/RTR.

Entretanto, com base nas diversas facilidades relativas a distribuição existentes em Java, é possível prever-se que a inclusão de distribuição em Java/RTR não será uma tarefa muito complexa, uma vez que tais facilidades são providas em Java através de bibliotecas de classes e que Java/RTR já dispõe de mecanismos para representação de *timeout* e *deadline*. Conjuntamente, estas classes e estes mecanismos constituem a base para o desenvolvimento de aplicações distribuídas. Em função disso, a inclusão de distribuição não exigirá qualquer alterações profundas de Java/RTR, podendo ser acomodada com base em ajustes na definição dos meta-objetos "manager" e do meta-objeto "scheduler"; contudo a confirmação desta previsão depende de uma análise profunda do suporte a distribuição oferecido por Java (o pacote Java.NET e os mecanismos RMI - *Remote Method Invocation* e *Object Serialization)*.

O uso de Java/RTR em ambientes distribuídos abertos também poderá ser considerado futuramente, podendo basear-se na extensão distribuída do modelo RTR descrita no capítulo anterior, ou então em uma das extensões tempo real de CORBA que atualmente se encontram em desenvolvimento. Em qualquer caso, deverá ser considerada a possibilidade de uso de ferramentas Java (tais como IDL/Java e Java ORB, atualmente em fase de testes) destinadas a integrar Java com outros ambientes e ferramentas baseados no padrão CORBA.

IV.3.4 - Implementação de Java/RTR

IV.3.4.1 - Introdução

Para implementarmos Java/RTR, optamos pelo uso de um pré-processador, denominado PP-Java/RTR, o qual fará a tradução de programas escritos em Java/RTR para programas Java equivalentes, mapeando as extensões introduzidas para código Java e mantendo intacto o código Java usado originalmente. O programa Java resultante deste pré-processamento deverá ser tratado como uma aplicação Java convencional; ou seja, deverá ser compilado pelo compilador "Javac" e executado pelo interpretador "java", em qualquer ambiente operacional para o qual a máquina virtual Java tenha sido portada.

Embora outras abordagens pudessem ser utilizadas na implementação de Java/RTR, tais como tradução direta para "Bytecodes" ou mesmo geração de código para um máquina virtual estendida, optamos por uma tradução para Java pelos seguintes motivos:

- Caráter experimental de Java/RTR;

- Menor esforço e tempo para sua implementação;

- Possibilidade de uso integral e direto dos API's e ferramentas de apoio disponíveis no ambiente de desenvolvimento Java;

- Manutenção das vantagens da portabilidade de Java.

Complementarmente, embora o PP-Java/RTR tenha como principal objetivo permitir a avaliação prática de Java/RTR e indiretamente do modelo RTR, sua implementação também será usada para avaliar a adequação de Java e de sua máquina virtual para o desenvolvimento de aplicações tempo real segundo o modelo RTR.

IV.3.4.2 - O pré-processador Java/RTR

O pré-processador Java/RTR (PP-Java/RTR) fará análise léxica e sintática dos programas a partir das especificações de Java/RTR. A especificação léxica é a mesma de Java [Sun 95a] à qual foi introduzido o item léxico "@", usado em Java/RTR para designar ativações assíncronas. Por outro lado, com relação a sintaxe, usaremos uma gramática livre de contexto (LALR(1)) que estende a gramática original de Java [Sun 95a] com as extensões introduzidas na seção anterior. A especificação formal da sintaxe destas extensões é apresentada no apêndice "A".

O PP-Java/RTR fará todas as verificações semânticas descritas na seção anterior e traduzirá as extensões introduzidas, mapeando-as para código Java equivalente; a tradução só será realizada quando o programa Java/RTR analisado estiver sintática e semanticamente (com relação as extensões utilizadas) correto. A semântica de Java não será analisada na fase de pré-processamento, devendo ser considerada posteriormente na compilação e execução do programa Java gerado.

Apresentamos a seguir uma descrição dos principais procedimentos do PP-Java/RTR com relação a verificação das extensões introduzidas e a tradução destas extensões para Java.

Declaração de Classes - Para cada classe/meta-classe Java/RTR analisada, será verificado se suas relações de herança estão de acordo com a especificação da linguagem, segundo a qual classes de um determinado tipo só podem estender classes deste mesmo tipo, com exceção de classes-base "Real-Time" (RTBC) que além de poderem estender outras classes RTBC, também podem estender classes convencionais Java (caracterizando-se neste caso como sendo RTBC especiais). A não verificação desta condição será considerada um erro de compilação. Por outro lado, se a classe em questão for uma meta-classe "Manager" (MMC), meta-classe "Scheduler" (SMC) ou meta-classe "Clock" (CMC) e não estiver estendendo nenhuma outra meta-classe de seu respectivo tipo, ela deverá, obrigatoriamente, implementar a interface padrão predefinida correspondente.

Os diferentes tipos de classes e meta-classes de Java/RTR serão traduzidos para classes convencionais Java, através da eliminação da opção Java/RTR que define a classe como sendo RTBC, MMC, SMC ou CMC; esta informação passará a ser uma constante do tipo string da classe gerada, cujo valor será, respectivamente "RTBC" (ou "SpecialRTBC"), "MMC", "SMC" ou "CMC". Esta informação será usada posteriormente pelo pré-processador,

para verificação da conformidade estrutural de programas Java/RTR com a especificação do modelo RTR e para tradução de outras extensões (tais como criação de meta-objetos e interação entre objetos). A obtenção desta informação será feita de forma introspectiva (ou seja, com base nas informações mantidas pelo suporte de Java), através dos recursos de reflexão disponíveis no pacote Java.Lang.Reflect do ambiente de desenvolvimento Java (versão 1.1).

Criação de Objetos - Sempre que um objeto for criado (através da operação new()) em um programa Java/RTR, o pré-processador verificará o tipo da classe da qual o objeto está sendo instanciado, e dependendo deste tipo fará um dos seguintes procedimentos:

- Se a classe for do tipo "MMC", "SMC", ou "CMC", um erro de compilação será acusado, uma vez que estas classes não podem ser instanciadas explicitamente;

- Se a classe for do tipo "RTBC", além da criação do objeto desejado, será criado adicionalmente um segundo objeto, que fará o papel de seu meta-objeto "manager", instanciado a partir da MMC correspondente a RTBC em questão; a não existência da classe MMC implicará em um erro de compilação;

- Se a classe for do tipo "SpecialRTBC", o procedimento será similar ao caso anterior, só que neste caso, a inexistência da MMC implicará na criação de um meta-obejto "manager" especial, instanciado a partir da classe pré-definida "SpecialMMC";

- Se a classe for convencional, nenhum procedimento adicional será realizado.

Criação dos meta-objetos "scheduler" e "clock" - Conforme a especificação de Java/RTR, o meta-objeto "Scheduler" e o meta-objeto "Clock" a serem usados na aplicação, serão criados implicitamente pelo PP-Java/RTR, quando da compilação do método main() da aplicação. Estes meta-objetos serão instanciados a partir, respectivamente, das meta-classes SMC e MMC informadas ao PP-Java/RTR quando de sua ativação. Caso as classes informadas não sejam do tipo devido, um erro de compilação será acusado; por outro lado se nada for informado, os meta-objetos "Scheduler" e "Clock" serão instanciados, respectivamente, a partir das meta-classes "StandardSMC" e "StandardCMC" predefinidas.

Declaração RT-Type - Para cada novo tipo de restrição temporal introduzido nas classes Java/RTR através da declaração **RT-Type**, serão verificadas as seguintes condições:

- se a classe em questão é uma RTBC; caso não seja, deverá ser acusado um erro de compilação;

- se a restrição temporal que está sendo introduzida já foi declarada na classe em questão; neste caso também será acusado erro de compilação;

- Se a meta-classe MMC correspondente possui um método que implementa a restrição temporal que está sendo declarada; esta verificação será feita de forma introspectiva através dos mecanismos de reflexão disponíveis em Java, sendo que a não existência do referido método também causará um erro de compilação.

A exigência de declaração de novos tipos de restrições temporais tem o objetivo de evitar que restrições temporais não declaradas ou não implementadas sejam eqüivocadamente associadas aos métodos da classe que está sendo compilada; adicionalmente, a declaração de tipos de restrições temporais permitirá a verificação da correta associação das restrições aos métodos do objeto, com relação a correspondência em número e tipo dos atributos.

Declaração de métodos em RTBC - Como especificado em Java/RTR, métodos de classes RTBC podem ter restrições temporais, manipuladores de exceção e categorias de controle associadas. Para tanto, será verificado inicialmente se os métodos aos quais estas informações estão sendo associadas, pertencem a uma RTBC, caso não pertençam será acusado um erro de compilação.

Para cada método de uma classe RTBC ao qual estas informações estejam sendo associadas, será criado um descritor contendo informações relativas a sua restrição temporal, seu manipulador de exceções e sua categoria de controle; este descritor será introduzido implicitamente na meta-classe "manager" (MMC) correspondente a classe RTBC na qual o método está sendo declarado (como sendo um elemento do *array* de descritores mantido pela MMC). Estas informações serão utilizadas no meta-processamento realizado quando estes métodos forem ativados, sendo que a ativação dos métodos que implementam as restrições temporais, os manipuladores de exceção e as categorias de controle (informados nestes descritores) serão realizados usando as facilidades reflexivas de Java.

Adicionalmente, caso uma restrição temporal do tipo "TT" seja associada a um determinado método, o PP-Java/RTR deverá gerar código para que este método seja ativado sempre que um objeto da classe em questão for criado.

Ativação de métodos em objetos-base "Real-Time" - Como especificado na seção IV.3.3.4, a partir de objetos-base "Real-Time" só podem ser ativados métodos de objetos convencionais ou de objetos-base "Real-Time". Os procedimentos semânticos e de tradução relativos a tais ativações, dependem dos seguintes fatores:

- Se a ativação é síncrona ou assíncrona;

- Se o método possui ou não restrição temporal associada;

- Se a ativação possui ou não um *timeout* associado.

No caso geral, independentemente dos fatores acima especificados, as ativações de métodos de objetos-base "Real-Time" serão tratadas reflexivamente, sendo convertidas implicitamente para ativações do método *ReceiveRequests()* do meta-objeto "Manager" correspondente ao objeto-base onde se encontra o método solicitado. As informações relativas ao método solicitado (identificador e parâmetros) serão passadas para o método *ReceiveRequests()* na forma de parâmetros reais. Por outro lado, ativações de métodos de objetos convencionais, serão mantidas intactas, a menos que sejam assíncronas ou possuam um *timeout* associado; nestes casos, os procedimentos relativos a tradução são similares ao caso de ativação de métodos de objetos-base "Real-Time", como descrito nos itens 2 e 3 a seguir.

Dependendo dos fatores acima mencionados, serão realizadas as seguintes verificações e procedimentos de tradução adicionais:

1 - Métodos com restrição temporal associada - Neste caso o comando de ativação deverá possuir um segundo conjunto de parâmetros reais (relativos aos atributos temporais da restrição em questão), devendo haver correspondência em número e tipo entre os valores informados e os atributos especificados na restrição temporal em questão. A não observação destas condições causará um erro de compilação. Por outro lado, os parâmetros temporais reais serão convertidos em parâmetros adicionais do método ReceiveRequests(). No caso do método ativado pertencer a um objeto convencional, a presença de parâmetros temporais na sua ativação, será considerada um erro de compilação.

2 - Ativações assíncronas - Toda chamada assíncrona será substituída pela criação de uma nova *thread*, instanciada a partir de uma classe auxiliar (que estende a classe *Thread* de Java), cujo método *Run()* conterá apenas a ativação do método a ser executado assincronamente. Este método poderá pertencer a um objeto convencional ou a um objeto-base "Real-Time", sendo que neste último caso a chamada será redirecionada para o método *ReceiveRequest()* do meta-objeto "Manager" correspondente ao objeto-base em questão.

3 - Ativações com *timeout* associado - Neste caso além da verificação da semântica associada a cláusula *timeout*, deverão ser gerados segmentos de código necessários para que o *timeout* especificado seja controlado. Entretanto, estes procedimentos só deverão ser incorporados ao PP-Java/RTR em uma futura versão de Java/RTR com suporte a distribuição.

Path-expression - Como especificado na seção anterior, as restrições de sincronização entre os métodos de um objeto-base podem ser expressas através de uma *path-expression* especificada no meta-objeto "Manager" correspondente. A análise e a tradução desta especificação, envolve os seguintes procedimentos do PP-Java/RTR:

- verificar se a classe na qual a *path-expression* está sendo especificada é ou não uma meta-classe "Manager";

- verificar a sintaxe da *path-expression* especificada, de acordo com gramática de Java/RTR;

- verificar se todos os operandos da *path-expression* são métodos do objeto-base em questão;

- gerar o autômato finito correspondente;

- introduzir na meta-classe "Manager" em questão, código relativo a declaração e inicialização de um *array*, representando a tabela de transições do autômato gerado.

Disciplina de programação - Além de realizar os procedimentos especificados nos itens anteriores, o PP-Java/RTR também terá a função de verificar sistematicamente alguns aspectos adicionais relativos a disciplina de programação necessária para que o comportamento desejado possa ser obtido. Dentre estes aspectos destacamos os seguintes:

- verificar se as restrições relativas a ativação de métodos presentes nos diferentes tipos de classes e meta-classes (RTBC, MMC, SMC e CMC) foram observadas pelo programador da aplicação;

- verificar se a proibição do uso dos mecanismos Java para criação e manipulação de "threads" foi observada na programação de classes do tipo RTBC.

IV.4 - Análise do tempo de execução de programas Java/RTR

A análise do tempo de execução de programas é uma tarefa extremamente complexa e a despeito das várias pesquisas realizadas e em desenvolvimento ([Pushner 89], [Park 91, 93], [Vrchoticky 94] e [Gustafsson 94], por exemplo), continua sendo uma área de pesquisa em aberto. Esta complexidade decorre da necessidade de se considerar diversos aspectos que vão desde o ambiente operacional (*hardware* e sistema operacional) no qual o programa será executado, até a linguagem de programação (incluindo-se aqui seu compilador e seu suporte de execução) na qual o programa será escrito.

Como visto na seção II.5, duas abordagens são tipicamente empregadas na obtenção do tempo de execução de um programa: uma estática, baseada em cálculo e outra dinâmica, baseada em medição. A abordagem baseada em cálculo tem como grande vantagem o fato de fornecer resultados relativos ao pior caso *(worstcase)*, que embora seja super-estimado (pessimista), oferece uma confiabilidade essencial para o caso de sistemas tempo real *hard*; por outro lado, a principal desvantagem é que ela só é viável mediante a proibição do uso de construções dinâmicas e/ou com tempo de execução imprevisível.

No outro extremo, a abordagem baseada em medição, embora não imponha restrições a nível das construções de linguagem, fornece apenas valores referentes ao caso médio (geralmente obtidos a partir de casos de testes), sendo portanto mais adequada para análise do comportamento temporal de sistemas tempo real *soft*.

Embora Java/RTR, destine-se prioritariamente a programação de aplicações tempo real *soft*, dependendo do ambiente operacional e de uma disciplina de programação adequada, pode ser possível o fornecimento de garantias dinâmicas para, pelo menos, algumas das tarefas que compõe uma aplicação; desta forma quanto mais confiável forem os tempos de execução das tarefas, maior será a possibilidade de que tais garantias sejam providas. Em função disto, estamos propondo uma abordagem baseada em cálculo (quando possível) e medição (nos caso onde o cálculo não é possível) e que permite o ajuste dinâmico dos tempos de execução obtidos via medição.

Esta abordagem justifica-se pelo fato de que, quanto maior for a confiabilidade dos tempos de execução das tarefas, melhores são as condições para que as violações das restrições temporais sejam antecipadamente detectadas e conseqüentemente maior será a disponibilidade de tempo para que as ações alternativas sejam executadas. Além disso, o uso de tempos mais confiáveis, favorece o controle de admissão de tarefas e também permite o estabelecimento de valores para os atributos das restrições temporais (tais como Período e *deadline*, por exemplo) de forma mais coerente.

A abordagem proposta engloba os seguintes passos:

1 - Cálculo do tempo de execução, quando possível;

2 - Medição do tempo de execução das tarefas para as quais o cálculo não é possível; e

3 - Ajuste dinâmico dos tempos de execução obtidos via medição.

Conjuntamente estes passos configuram uma abordagem mista, onde algumas das vantagens (como por exemplo obtenção do *worstcase*) da abordagem baseada em cálculo podem ser obtidas para determinadas tarefas que sigam uma determinada disciplina de programação (comentada mais adiante) e algumas das desvantagens da abordagem baseada em medição (como por exemplo a baixa confiabilidade dos valores obtidos) podem ser reduzidas através do ajuste dinâmico dos tempos estimados a *priori*.

O uso de abordagens mistas não é novo, e em particular, nossa abordagem é inspirada na proposta de [Nilsen 96a] para análise do tempo de execução de programas RT-Java. Entretanto, enquanto [Nilsen 96a] propõe que cálculo e medição sejam realizados durante a execução do sistema (mais especificamente na fase de negociação de recursos, a qual antecede a aceitação ou não da tarefa na carga atual do sistema) e necessita de uma máquina virtual Java especial, nossa proposta deve permitir que o cálculo seja feito em tempo de compilação ou pelo menos antes da execução (portanto sem a necessidade de envolvimento da máquina virtual Java) e que as medições sejam realizadas também em uma fase anterior a operação do

sistema (possivelmente na fase de testes). Desta forma elimina-se a interferência dos procedimentos de cálculo e reduz-se a interferência do processo de medição no tempo de execução do sistema; por outro lado, a interferência do ajuste dinâmico dos tempos de execução é mínima, na medida em que nenhuma execução extra será necessária (os novos valores serão obtidos a partir da medição de execuções previstas no fluxo normal do sistema).

Por outro lado, esta proposta exigirá um esforço extra no projeto e implementação de ferramentas que sistematizem (pelo menos parcialmente) os processos de cálculo e medição, enquanto que na proposta de [Nilsen 96a] este esforço é mínimo, uma vez que a estrutura da máquina virtual Java é aproveitada.

IV.4.1 - Considerações gerais sobre a abordagem proposta

No que segue, são descritas algumas idéias preliminares relativas a abordagem proposta para obtenção do tempo de execução de programas Java/RTR.

Cálculo do tempo de execução - Como Java/RTR não restringe nem sintática nem semanticamente o uso de construções dinâmicas e/ou com tempo de execução imprevisível em seus programas, o cálculo do tempo de execução dos métodos (tarefas) dos objetos que compõem uma aplicação só pode ser realizado através de uma disciplina de programação na qual estas construções não sejam utilizadas. As regras que compõem esta disciplina são, em princípio, aquelas estabelecidas em [Pushner 89] e adaptadas em [Nilsen 96a] para o caso específico de cálculo do tempo de execução de programas RT-Java (como descrito, sucintamente na seção II.5.3.4).

A verificação desta disciplina (composta por um conjunto de regras que proíbem ou restringem o uso de construções tais como *recursão* e *loop's* ilimitados) deverá ser feita sistematicamente pela ferramenta responsável pelo procedimento de cálculo do tempo de execução, sendo que a violação de qualquer das restrições impostas causará a interrupção imediata do processo de cálculo e o método cujo tempo estava sendo calculado deverá ser declarado como não analisável quanto ao cálculo do seu tempo de execução.

A ferramenta responsável pelo cálculo deverá operar sobre uma representação do programa Java gerado pelo pré-processador Java/RTR, disposta na forma de um grafo de dependências entre os métodos que compõem a aplicação. O processo de cálculo deve ser iniciado a partir da análise de métodos sem dependências (ou seja, métodos que não ativam outros), passando então para a análise dos métodos que dependem apenas de métodos já analisados e assim sucessivamente até que todos os métodos sejam analisados. A constatação de que um determinado método não pode ter seu tempo de execução calculado, faz com que todos os demais métodos que dependam dele (direta ou indiretamente) não sejam analisáveis.

Na prática, além da verificação das restrições impostas para realização do cálculo do tempo de execução, duas situações especiais devem ser consideradas: uso de métodos das classes padrão de Java e tempo gasto no meta-processamento.

Com relação aos métodos padrão de Java, a idéia é que tais métodos não sejam analisados em cada aplicação, mesmo porque muitos deles violam a disciplina de programação necessária para a realização do cálculo; assim sendo a solução poderá ser o uso de uma base de dados contendo o tempo de execução de tais métodos, cujo valor seria usado como constante no cálculo do tempo de execução dos métodos da aplicação que o utilizam. Para tanto, o tempo de execução destes métodos seria obtido através de cálculo (quando possível) ou de medições (usando-se uma bateria de testes significativa, visando a obtenção de

valores mais confiáveis), realizados uma única vez para cada ambiente onde Java/RTR estiver instalada.

Por outro lado, o tempo gasto no meta-processamento (relativo ao controle dos métodos dos objetos-base da aplicação), constitui-se em parte do tempo de execução total dos métodos dos objetos-base e portanto deve ser computado no cálculo do tempo de execução destes métodos; assim sendo, como no caso anterior, a idéia é que o tempo de execução destes métodos seja previamente obtido (via medição ou quando possível via cálculo) e registrado na referida base de dados, podendo ser utilizado diretamente (como um valor constante) no cálculo do tempo de execução dos métodos dos objetos-base. Contudo convém salientar que somente pequena parte do tempo gasto no meta-processamento é *overhead* devido a reflexão, pois a maior parte deste tempo é gasta na execução de funções de controle, as quais são necessárias mesmo em ambientes não reflexivos.

Medição - A proposta básica relativa a medição, é a construção de uma ferramenta que permita, através de uma pré-execução, medir o tempo de execução de todos os métodos que compõem uma aplicação; tal ferramenta deverá usar o mesmo grafo de dependências de métodos e os mesmos procedimentos de análise utilizados para o cálculo. Além disso, da mesma forma que no caso anterior, deverá existir uma base de dados com o valor obtido a partir de medições relativas aos métodos mais comumente utilizados, tais como os métodos das classes padrões de Java e os métodos das meta-classes padrões de Java/RTR. O uso desta base (que poderá ser a mesma usada pela ferramenta de cálculo - neste caso aumentando a confiabilidade dos valores finais obtidos) não só agilizará o processo de medição, como também poderá ser usado como base para o estabelecimento dos valores de outros atributos das restrições temporais (quando estes não forem inerentes a própria aplicação).

Ajuste dinâmico do tempo de execução - este passo da metodologia proposta tem como objetivo aumentar a confiabilidade nos valores que representam os tempos de execução dos métodos obtidos via medição. Para realização desta tarefa, a idéia inicial é usar o potencial do modelo RTR (conforme descrito na seção III.5), embutindo os procedimentos de medição e ajuste dinâmico nos procedimentos do meta-objeto manager; desta forma, o ajuste seria mais uma função de controle realizado reflexivamente.

Embora este passo possa contribuir para o incremento da confiabilidade dos tempos de execução obtidos, ele interfere no tempo total de execução (da aplicação em geral e de um determinado método em particular) e contribui para o aumento do *overhead* devido a reflexão; visando reduzir esta interferência, o processo de medição e ajuste poderá ser realizado de forma seletiva, por exemplo, envolvendo apenas métodos com determinadas restrições temporais associadas (como periodicidade, por exemplo) ou um conjunto de métodos previamente selecionados (através da cláusula "**categoria**", por exemplo).

IV.5 - Conclusões

Neste capítulo apresentamos a linguagem Java/RTR, uma extensão de Java que incorpora explicitamente a estrutura e o comportamento do modelo RTR. Além das extensões propriamente ditas, também foi apresentada uma especificação preliminar do pré-processador que fará a tradução de programas Java/RTR para programas Java eqüivalentes e proposta uma abordagem para análise do tempo de execução dos programas Java/RTR.

A linguagem Java/RTR pode ser vista como um veículo para realização prática da filosofia de programação do modelo RTR, a qual herda as vantagens do modelo RTR ao mesmo tempo em que preserva as principais características de Java.

Comparativamente com outras linguagens tempo real orientadas a objetos, Java/RTR mostra-se mais expressiva e flexível e apresenta mais facilidades para o gerenciamento da complexidade inerente às aplicações tempo real; além disso, o esquema reflexivo usado para representação e controle dos aspectos temporais, aliado a independência de arquitetura de Java, dão a Java/RTR uma independência de suporte de execução e de ambiente operacional não encontrada em nenhuma outra linguagem tempo real.

Por outro lado, a utilização de um esquema de concorrência/sincronização no nível-base diferente do esquema nativo de Java usado no meta-nível, afeta a ortogonalidade de Java/RTR, restringe o uso de facilidades Java e reduz a possibilidade de reuso de classes convencionais Java; contudo, esta questão deverá ser revista em uma futura versão de Java/RTR, no sentido de adaptar os mecanismos nativos de Java às necessidades impostas pela abordagem RTR. Um segundo problema de Java/RTR é a questão da previsibilidade, cuja obtenção é dificultada em função da flexibilidade provida e da manutenção de construções (*loop's* ilimitados, por exemplo) e mecanismos (coletor de lixo, por exemplo) imprevisíveis de Java; entretanto, esta dificuldade pode vir a ser contornada através da substituição de tais mecanismos e do estabelecimento de uma disciplina de programação que proíba, ou pelo menos limite, o uso de construções imprevisíveis.

Capítulo V - Conclusões

Neste trabalho foi proposto um modelo de programação concorrente, orientado a objetos e reflexivo para programação de aplicações tempo real. O modelo proposto, denominado Modelo RTR, não só permite a representação e o controle de restrições temporais de forma explícita, flexível e natural, como também contribui para a redução de vários problemas encontrados atualmente na programação de sistemas tempo real. A potencialidade e a expressividade do modelo proposto foram demonstradas através da identificação de várias situações tempo real representáveis pelo modelo RTR e da representação das questões de sincronização presentes em aplicações multimídia. Além disso, a extensibilidade do modelo bem como sua capacidade de adaptação a diferentes ambientes operacionais pôde ser constatada através da proposição e da implementação de uma extensão distribuída do modelo RTR para ambiente abertos baseada na arquitetura CORBA.

Adicionalmente, para facilitar o uso prático do modelo RTR, também foi proposta uma linguagem de programação, denominada Java/RTR, a qual estende a linguagem Java com suporte para reflexão computacional e para representação de aspectos temporais, permitindo que a estrutura e o comportamento do modelo RTR sejam usados explicitamente na programação de aplicações tempo real.

- **Considerações sobre o modelo e a linguagem propostos**

De forma geral, as contribuições deste trabalho são a definição de um modelo de programação tempo real e a especificação de uma linguagem de programação que implementa este modelo. Do nosso ponto de vista, o modelo e a linguagem propostos representam um avanço à atividade de programação tempo real, na medida em que reduzem algumas das dificuldades encontradas atualmente no desenvolvimento de sistemas tempo real programados segundo modelos e linguagens existentes.

Além disso, destacamos as seguintes contribuições decorrentes do trabalho realizado:

- sistematização da implementação de restrições temporais presentes nas relações de sincronização de aplicações multimídia, a partir do modelo de especificação baseado em intervalos;

- integração do modelo RTR com CORBA, permitindo que sua filosofia seja também empregada na programação de aplicações tempo real em ambientes distribuídos abertos;

- integração do modelo RTR com a linguagem Java, permitindo que as facilidades desta linguagem sejam também empregadas na programação de aplicações tempo real.

As principais vantagens e as limitações do modelo e da linguagem propostos, são descritos a seguir.

Modelo RTR - Dentre as principais vantagens do modelo proposto, destacamos sua expressividade, sua flexibilidade, sua abrangência e sua efetividade para representação e controle dos aspectos temporais das aplicações, além de sua independência de suportes e de ambientes operacionais específicos. Estas vantagens são decorrentes dos seguintes aspectos:

- facilidades relativas a estruturação de sistemas tempo real, facilitando o entendimento e conseqüentemente o gerenciamento da complexidade destes sistemas;

- flexibilidade para definição e uso de novos tipos de restrição temporal e algoritmos de escalonamento, os quais são definidos e controlados no meta-nível da aplicação de acordo com as especificidades de diferentes classes de aplicação;

- capacidade de reuso e manutenção de objetos-base e meta-objetos, decorrente da separação explícita entre questões funcionais e questões de controle;

- facilidade para adaptação de novas técnicas e mecanismos decorrentes da evolução da teoria de tempo real em geral e da teoria de escalonamento em particular;

- extensibilidade, decorrente da capacidade do modelo RTR para acomodar o controle e o gerenciamento de questões correlatas a tempo real, tais como distribuição e tolerância a faltas por exemplo.

Por outro lado, dentre as limitações do modelo proposto destacamos que:

- previsibilidade não é inerente : sua obtenção depende de implementações particulares e dos ambientes operacionais sobre os quais estas implementações vierem a ser realizadas. Entretanto, no caso geral, como em qualquer modelo onde a flexibilidade é privilegiada, no modelo RTR a obtenção de previsibilidade também é dificultada;

- o alto grau de abstração do modelo RTR, embora flexibilize sua utilização em diferentes classes de aplicação sob diferentes ambientes operacionais, exige que vários aspectos sejam adaptados e/ou redefinidos em função dos objetivos e da disponibilidade de recursos de cada implementação particular do modelo;

- o esquema reflexivo utilizado impõe um certo grau de *overhead* que, embora seja aceitável e gerenciável no caso geral, pode se constituir em um fator de limitação em situações particulares onde a taxa de utilização de recursos de processamento seja extremamente alta;

Linguagem Java/RTR - Como resultado da integração entre o modelo RTR e a linguagem Java, Java/RTR permite a representação e o controle das questões temporais ao mesmo tempo em que preserva as facilidades de Java. Dentre as principais vantagens decorrentes desta integração, destacamos:

- expressividade e flexibilidade para representação dos aspectos temporais;

- capacidade de reuso e manutenção, mesmo na presença de aspectos temporais;

- redução da complexidade na estruturação e no entendimento de sistemas tempo real, decorrente da separação explícita entre questões funcionais e de controle;

- independência de ambiente operacional, decorrente do tratamento reflexivo das questões temporais e da independência de arquitetura de Java;

- redução do "gap" semântico entre projeto e implementação de sistemas tempo real, em função da incorporação explícita do modelo RTR a nível de linguagem;

Além disso, comparativamente com outras linguagens tempo real, Java/RTR é menos complexa na programação e mais efetiva no controle da concorrência dos sistemas tempo real; isto deve-se ao tratamento reflexivo das questões de concorrência e sincronização, realizado de forma integrada ao controle das questões temporais.

Por outro lado, a integração entre o modelo RTR e a linguagem Java, resultou em algumas limitações na linguagem Java/RTR, dentre as quais destacamos:

- o esquema de concorrência/sincronização de Java/RTR; embora viável e compatível com o modelo RTR, este esquema fere a ortogonalidade da linguagem;

- dificuldade de reutilização de classes Java (que utilizam facilidades multithreading) desenvolvidas fora do escopo de Java/RTR;

- a imprevisibilidade decorrente da presença de mecanismos (*garbage colector*, por exemplo) e construções (*loop's* ilimitados e recursões, por exemplo) com tempo de execução imprevisível. Embora esta limitação seja comum a todas as linguagens tempo real orientadas a objetos que oferecem algum grau de flexibilidade, pretendemos minimizá-la em uma futura versão de Java/RTR através da sistematização de uma disciplina de programação.

- a não consideração nesta versão de Java/RTR do potencial de Java relativo a distribuição e ao uso de *Applets* (programas Java embutidos em *Browsers*);

• **Perspectivas**

Dando continuidade ao trabalho aqui apresentado, estão previstas várias atividades relacionadas ao modelo, a linguagem e a utilização de ambos.

Com relação a Java/RTR, a curto prazo pretendemos implementar o seu pré-processador, detalhar a abordagem proposta para análise do tempo de execução de programas Java/RTR e implementar as ferramentas previstas nesta abordagem. A médio prazo, pretendemos propor uma nova versão de Java/RTR com o objetivo de resolver as limitações identificadas na presente versão.

Quanto ao modeloRTR, pretendemos estendê-lo estruturalmente através da introdução de meta-objetos especializados para classes de aplicações específicas, tais como trabalho cooperativo, sistemas tolerantes a falta e sistemas multimídia, visando o tratamento reflexivo de questões como coordenação [Fritske 97], tolerância a faltas [Fraga 97b] e qualidade de serviço.

No que diz respeito a utilização da abordagem proposta, as perspectivas são as seguintes:

- validação da abordagem como um todo e de Java/RTR em particular, a partir de sua utilização prática na programação de sistemas tempo real mais complexos;

- proposição de uma metodologia completa para desenvolvimento de sistemas tempo real baseada na filosofia de programação introduzida pelo modelo RTR. Esta metodologia deverá, idealmente, ser suportada por um ambiente de desenvolvimento composto por diversas ferramentas de apoio que permitam a sistematização de vários aspectos envolvidos no processo de desenvolvimento de sistemas tempo real.

- Criação de um ambiente fundamentado na abordagem proposta para ser utilizado como um laboratório para análise e validação de novas facilidades decorrentes da evolução da teoria tempo real, tais como novos tipos de restrições temporais e novas abordagens de escalonamento;

- integração do trabalho proposto com outros trabalhos desenvolvidos no âmbito do LCMI/DAS, como por exemplo a implementação da abordagem de escalonamento baseada em computação imprecisa para ambientes distribuídos proposta em [Oliveira 97].

REFERÊNCIAS BIBLIOGRÁFICAS

[Allen 83] - Allen, J. F. "Maintaining Knowledge about Temporal Intervals", Communications ACM, vol. 26, n. 11, pp. 832-843, november, 1993.

[ASAP 94] - "Um Ambiente para Suporte de Aplicações Distribuídas Baseado em Objetos", Projeto CNPq/PROTEM II, Campinas, março, 1994.

[Blakowski 96] - Blakowski, G. and Steinmetz, R. "A Media Synchronization Survey : Reference Model, Specification, and Case Studies". IEEE Journal in Selected Areas in Communications, 14(1) : 5-35, 1996.

[Berryman 93] - Berryman, S. J. "Modelling and Evaluating Time Constraints in RealTime Systems", Thesis, Lancaster University, março 1993.

[Bihari 92] - Bihari, T. E. and Gonipath, P. "Object-Oriented Real-Time Systems: Concepts and Examples", Computer, december, 1992.

[Bosh 97] - Bosh, J. and Molin, P. "A Model for a Flexible and Predictable Object-Oriented Real-Time Systems", WORDS'97, Newport Beach, Ca, USA, february, 1997.

[Burns 90] - Burns, A. e Audsley, N. - "Real-Time.Systems Scheduling", University of York, 1990.

[Burns 96a] - Burns, A. "Concurrent and Real-Time Programming in ADA95", Notes of the talk presented at WRTP'96, Gramado, RS, Brasil, november 4-6, 1996.

[Burns 96b] - Burns, A. and Wellings, A. "Real-Time Systems and Programming Languages", second edition, Addison-Wesley, 1996.

[Campbell 74] - Campbell, R. H. e Habermann, A. N. "The Specification of Process Synchronization by Path-Expressions", In Operating Systems, Kaiser, C. (Ed.) Berlin: Springer-Verlag, 1974.

[Chiba 93a] - Chiba, S. "Open C++ Programmer's Guide", Technical Report 93-3, Departament of Information Science, University of Tokio, 1993.

[Chiba 93b] - Chiba, S. e Masuda, T. "Designing an Extensible Distributed Language with a Meta-Level Architecture", In Proceedings of 7th European Conference on Object-Oriented Programming (ECOOP'93), pp. 482-501, Kaiserslautem, julho 1993.

[Chiba 95] - Chiba, S. 'A Metaobject Protocol for C++", in OOPSLA'95 Proceedings, 1995.

[Curtis 97] - Curtis, D. "Java, RMI and CORBA - A White Paper", Object Management Group, http://www.omg.org/news/wpjava.htm, 1997.

[Dourish 96] Dourish, P. "Open Implementation and Flexibility in CSCW Toolkits", Thesis, University of London, june, 1996.

[Ellis 93] - Ellis, M. A., Stroustrup, B. "C++ - Manual de Referência Comentado", Editora Campus, 1993.

[Eriksson 94] - Eriksson, C. "An Object-Oriented Framework for the Design of Hard Real-Time Systems - A Study Focused on RealTimeTalk", Licentiate Thesis, Royal Institute of Technology, Suécia, 1994.

[Fabre 95] - Fabre, J-C. et al. "Implementing Fault Tolerant Applications using Reflective Object-Oriented Programming", 25th FTCS, Pasadena, CA-USA, pp. 489-498, june, 1995.

[Fabre 96] - Fabre, J-C. and Pérennou, T. "FRIENDS: A Flexible Architecture for Implementing Fault Tolernat and Secure Distributed Applications", 2nd. EDCC, Taormina, Italy, october, 1996.

[Fraga 95] - Fraga, J., Farines, J.M., Furtado, O.J.V. e Siqueira, F. "A Programming Model for Real-Time Applications in Open Distributed Systems", 5th Workshop on Future Trends in Distributed Computing Systems, Cheju Island, Republic of Korea, agosto 1995.

[Fraga 96] Fraga, J., Farines, J-M., Furtado O. and Siqueira F. "Programação de Aplicações Distribuidas Tempo-Real em Ambientes Abertos", SEMISH'96 , Recife, PE, agosto, 1996.

[Fraga 97a] - Fraga, J., Farines, J-M. and Furtado, O. "RTR Model : An Approach for Dealing with Real-Time Programming in Open Distributed Systems", WORDS'97, Newport Beach, Ca, USA. February 5-7, 1997.

[Fraga 97b] - Fraga, J., et al. "Implementing Replicated Services in Open Systems Using a Reflective Approach", Proceedings of ISADS'97, pp. 273-282, Berlin, Germany, april 1997.

[Fritzke 97] - Fritzke Jr., U. e Farines, J-M. "Modelagem e Implementação de Aplicações de Computação Cooperativas em Ambientes Distribuídos Heterogêneos"; SEMISH'97, Brasília, DF, Brasil, agosto 1997.

[Furtado 95] - Furtado, O.J.V. "Um Modelo e uma Linguagem para Aplicações Tempo Real", Exame de Qualificação para Doutorado, PGEEL–UFSC, Outubro 1995.

[Furtado 96a] - Furtado, O., Siqueira, F., Fraga, J. and Farines, J-M. "A Reflective Model for Real-Time Applications in Open Distributed Systems". In: WRTP'96 Proceedings, Gramado, RS, Brazil, November 1996.

[Furtado 96b] - Furtado, O. and Farines, J-M. "Java/RTR - Uma Linguagem Reflexiva para Programação de Aplicações Tempo-Real". I Simpósio Brasileiro de Linguagens de Programação (SBLP'96), Belo Horizonte, MG, Brasil , setembro 1996.

[Furtado 97] - Furtado, O., Farines, J-M. e Fraga, J. "Explorando a Potencialidade e a Expressividade do Modelo Reflexivo Tempo Real RTR na Programação de Aplicações Tempo Real ", XXIII Conferência Latinoamericana de Informática, Vol. II, pp. 579-588, Valparaiso, Chile, novembro 1997.

[Gehani 91] - Gehani, N. e Ramamritham, K. "Real-Time Concurrent C : A Language for Programming Dynamic Real-Time Systems", The Journal of Real-Time Systems, n. 3, pp 377-405, 1991.

[Gonçalves 94] - Gonçalves, C. Jr. "Objetos Distribuídos", Dissertação de Mestrado, UNICAMP, Campinas, agosto 1994.

[Gustafsson 94] - Gustafsson, J. "Calculation of Execution Times in Object-Oriented Real-Time Software - A study Focused on RealTimeTalk", Licentiate Thesis, Royal Institute of Technology, Suécia, 1994.

[Halang 90] - Halang, W. A. e Stoyenko, A. D. "Comparative Evaluation of High-Level Real-Time Programming Languages", J. Time-Critical Comp. Systems, n. 2, pp 365-382, 1990.

[Harnon 94] - Harnon, M. G. et al. "A Retargetable Technique for Predicting Execution Time of Code Segments", Real-Time Systems, vol. 7, pp. 157-182, 1994.

[Honda 94] - Honda, Y. e Tokoro, M. "Reflection and Time-Dependent Computing : Experiences with the R2 Architecture", Technical Report, SONY C. S. Laboratory Inc., Tokio, julho 1994.

[Horn 92] - Horn, F. and Stefani, J. B. "On Programming and Supporting Multimedia Object Synchronization", The Computer Journal, vol. 36, n. 1, 1993.g

[Ishikawa 90] - Ishikawa, Y., Tokuda, H. e Mercer, C.W. "Object-Oriented Real-Time Language Design : Constructs for Timing Constraints", ECOOP/OOPSLA'90, pp 289-298, outubro 1990.

[Ishikawa 92] - Ishikawa, Y., Tokuda, H. e Mercer, C.W. "An Object-Oriented Real-Time Programmming Language", Computer, pp 66-73, outubro 1992.

[Karmouch 93] - Karmouch, A. "Multimedia Distributed Cooperative System", Computer Communications, Vol. 16, No. 9, September 1993.

[Kenny 91] - Kenny, K. B. e Lin, K. J. "Building Flexible Real-Time Systems Using the FLEX Language", Computer, pp 70-78, maio 1991.

[Kiczales 91] - Kiczales, G. et. al. "The Art of the Meta-Object Protocol", The MIT Press, 1991.

[Kim 93] - Kim, K.H. e Barcellar, L. F. "A Real-Time Object Model: A Step toward an Integrated Methodology for Engineering Complex Dependable Systems", Proceedings of CSESAW'93, julho 1993.

[Kim 94a] - Kim, H. K. e Kopetz, H. "A Real-Time Object Model RTO.k and an Experimental Investigation of its Potentials", COMPSAC'94, novembro 1994.

[Kim 94b] - Kim, H. K. et al. "Distinguishing Features and potentials Roles of the RTO.k Object Model", Proceedings of WORDS'94, outubro 1994.

[Kim 96] - Kim, K. et al. "Support for RTO.k Objects Structured Programming in C++", 21st. IFAC/IFIP Workshop on Real-Time Programming - WRTP'96, Gramado, RS, Brasil, november, 1996.

[Kopetz 91] - Kopetz, H. et al. "The design of Real-Time Systems: From Specification to Implementation and Verification", Software Engineering Journal, may, 1991.

[Kopetz 93] - Kopetz, H. et al. "Real-Time System Development : The Programming Model of MARS", Proceedings of ISADS'93, Kawasaki - Japan, pp 290-299, abril 1993.

[Lea 96] - Lea D. "Concurrent Programming in Java. Design, Principles and Patterns", Addison Wesley, october, 1996.

[Li 94a] - Li, G. "Distributing Real-Time Objects: Some Early Experiences", Project ANSA Phase III, APM.1231.00.01, External Paper, maio 1994.

[Li 94b] - Li, G. " Distributed Real-Time Objects : the ANSA Approach", In: WORDS'94 Proceedings, IEEE Pub, september, 1994.

[Lin 88] - Lin, K. J. "Expressing and Manutaining Timing Constraint in FLEX", Proceedings of the IEEE Real-Time Systems Symposium, pp 96-105, 1988.

[Lin 91] - Lin, K.-J., Liu, J.W.S., Kenny, K.B. and Natarajan, S. "Flex: A Language for Programming Flexible Real-Time Systems", chapter 10 of "Foundations of Real-Time Computing: Formal Specifications and Methods", edited by Tilborg, A. and Koob, G.. Kluwer Academic Publishers, USA, 1991.

[Lisbôa 96] - Lisbôa, M. L. B. e Rubira, C. M. F. "Técnicas de Orientação a Objetos para Tolerância a Falhas", I SBLP, pp.385-398, Belo Horizonte, MG, setembro de 1996.

[Little 90] - Little, T. D. C. and Ghafoor, A. "Synchronization and Storage Models for Multimedia Objects", IEEE Journal on Selected areas in Communication, vol. 8, n. 3, april, 1990.

[Little 94] - Little, T. D. C. "Time-Based Media Representation and Dellevery", in "Multimedia Systems", edited by Buford, J. F. K., ACM Press, SIGGRAPH Series, NY, 1994.

[Liu 73] - Liu, C. L. e Layland, J. W. "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", JACM, vol. 20, n.1, pp 46-61, janeiro 1973.

[Maes 87] - Maes, P. "Concepts and Experiments in Computational Reflection", OOPSLA'87 Proceedings, pp. 147-156, outubro 1987.

[Masuhara 92] - Masuhara, H. et al. "Object-Oriented Reflective Languages can be Implemented Efficiently", OOPSLA'92, pp. 127-144, 1992.

[Meyer 88] - Meyer, B. "Object-Oriented Software Construction", Prentice Hall International, 1988.

[Mitchell 96] - Mitchell, S. "TAO - A Model for the Integration of Concurrency and Synchronization in Object-Oriented Programming", PhD. Thesis, University of York, UK, march, 1996.

[Mitchell 97] - Mitchell, S. E., Burns, A. and Wellings, A. J. "Developing a Real-Time Metaobject Protocol", WORDS'97, Newport Beach, California, USA, february 5-7, 1997.

[Mok 89] - Mok, A. et al. "Evaluating Target Execution Time Bounds of programs by Annotations", Proceedings of 6. IEEE Workshop on Real-Time Operating Systems and Software, pp 74-80, maio 1989.

[Nilsen 95] - Nilsen, K. "Issues in the Design and Implementation of Real-Time Java", Technical Report, Iowa State University, USA november, 1995.

[Nilsen 95a] -Nilsen, K. and Rygg, B. "Worst-Case Execution Time Analysis on Modern Processors", in SIGPLAN'95, 1995.

[Nilsen 96a] - Nilsen, K. "Real-Time Java (draft 1.1)", Technical Report, Iowa State University, USA, february, 1996.

[Nilsen 96b] - Nilsen, K. "Embedded Real-Time Development in the Java Language", Iowa State University, USA, 1996.

[Nilsen 96c] - Nilsen, K. "PERC up your Java", Newmonics Inc., (www.newmonics.com), Ames, Iowa, USA, 1996.

[Nirkhe 93] - Nirkhe, V. e Pugh, W. "A Partial Evaluator for the MARUTI Hard Real-Time System", Real-Time System, vol. 5, pp. 13-30, 1993.

[Nishida 96] - Nishida, D. "Estudo e implementação do Modelo Reflexivo Tempo Real RTR sobre a Linguagem Java", Dissertação de Mestrado - LCMI/EEL/UFSC, Florianópolis, SC, Brasil, dezembro 1996.

[Nishida 97] - Nishida, D., Furtado, O., Fraga, J. e Farines, J-M. "Um Protótipo do Modelo Reflexivo Tempo Real RTR sobre a Linguagem Java", XXIII Conferência Latinoamericana de Informática, Vol. II, pp. 589-598, Valparaiso, Chile, novembro 1997.

[Oliveira 97] - Oliveira, R. S. "Escalonamento de Tarefas Imprecisas em Ambiente Distribuído", tese de Doutorado, LCMI, UFSC, abril 1997.

[OMG 96a] - "CORBA 2.0 Specification", OMG Technical Document PTC/96-03-04, USA. 1996.

[OMG 96b] - OMG - CORBA, "ORB and Objects Services RFI2 (Realtime)", http://www.omg.org/library/schedule/Realtime-RFI.htm, october, 1996.

[OMG 96c] - OMG - CORBA, "Meta Object Facility RFP", Document CF/96-05-02, http://www.omg.org/library/schedule/CF_RFP5.htm, 1996.

[Park 91] - Park, C. Y. e Shaw, A. C. "Experiments with a Program Timing Tool Based on a Source Level Timing Schema", IEEE Computer, pp. 48-56, maio 1991.

[Park 93] - Park, C. Y. "Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths", Real-Time Systems, vol.5, pp. 31-62, 1993.

[PMC 95] - "ORBeline User's Manual and ORBeline Reference Manual", PMC - Post Modern Computing Inc., USA, 1995.

[Pospischil 92] - Pospischil, G. et. al. "Developing Real-Time Tasks with Predictable Timing", IEEE Software, pp 35-44, setembro 1992.

[Pushner 89] - Pushner, P. e Koza, CH. "Calculating the Maximum Execution Time of Real-Time Programs", Journal of Real-Time Systems, vol. 1, pp. 159-176, 1989.

[Rubira 97] - Rubira, C. M. F., Correa, S. L. e Buzato, L. B. "Um Framework Orientaado a Objetos Reflexivo para a construção de Software Tolerante a Falhas", VII SCTF, Campina Grande, Pb, julho 1997.

[Ruf 93] - Ruf, E. "Partial Evaluation in Reflective Systems Implementation", in Proceedings OOPSLA'93, Workshop on Reflection and Metalevel Architectures, 1993.

[Siqueira 96] - Siqueira, F. "Programação de Aplicações com Requisitos Temporais em Sistemas Abertos: O modelo de Objetos Reflexivo Tempo Real Distribuído", Dissertação de Mestrado , LCMI/EEL/UFSC, Florianópolis, SC, Brasil, Junho 1996.

[Siqueira 96a] - Siqueira, F., Furtado, O., Fraga, J. e Farines, J-M. "Implementação Distribuída de um Modelo Reflexivo Tempo Real", I Workshop de Sistemas Distribuídos (WOSID), Salvador, Ba, maio 1996.

[Stankovic 88] - Stankovic, J. A. "Misconceptions About Real-Time Computing", IEEE Computer, vol. 21, n. 10, outubro 1988.

[Stankovic 93] - Stankovic, J. A. "Reflective Real-Time Systems", Technical Report, DCS, University of Massachusetts, june, 1993.

[Stankovic 94] - Stankovic, J. A. e Ramamritham, K."Scheduling Algorithms and Operating Systems Support for Real-Time Systems", Proceedings of the IEEE, vol. 82, n. 1, janeiro, pp 55-67, 1994.

[Stankovic 96] - Stankovic, J. A., et al., "Strategic Directions in Real-Time and Embedded Systems", ACM Computing Surveys, vol 28, n. 4, pp. 751-763, december, 1996.

[Stoyenko 86] - Stoyenko, A.D. e Kligerman, E. "Real-Time EUCLID : A Language for Reliable Real-Time Systems", IEEE Transactions in Software Engineering, vol. SE-12, n. 9, pp 941-949, setembro 1986.

[Stoyenko 91] - Stoyenko, A. D., Hamacher, V. C. e Holt, R. C. "Analyzing Hard Real-Time Programs for Guaranteed Schedulability", IEEE Transactions on Software Egineering, vol. 17, n. 8, pp 737- 750, agosto 1991.

[Stoyenko 92] - Stoyenko, A. D. "The Evolution and State of-the-Art of Real-Time Languages", The Journal of Systems and Software, pp 61-84, abril 1992.

[Stoyenko 94] - Stoyenko, A. D. e Baker, T. P. "Real-Time Schedulability Analyzable Mechanisms in ADA9X", Proceedings of the IEEE, vol. 82, n. 1, pp 95-107, janeiro 1994.

[Sun 95a] - Sun Microsystems Inc. "The Java Language Specification - versao 1.0 beta", Mountain View, CA, http://www.javasoft.com/docs/books/jls/index.html, 1995.

[Sun 95b] - Sun Microsystems Inc. "The Java Language Tutorial", Mountain View, CA, http://www.javasoft.com/docs/books/Tutorial/index.html, 1995.

[Sun 95c] - Sun Microsystems Inc. "The Java Language Environment - A White Paper", Mountain View, CA, 1995.

[Sun 96a] - Sun Microsystems Inc."Java remote Method Invocation Specification - revision 0.9", Mountain View, CA, may 1996.

[Sun 96b] - Sun Microsystems Inc. "JavaCore Reflection - API and Specification", Mountain View, CA, september, 1996.

[Sun 97] - Sun Microsystem Inc. "Java IDL Guide", http://www.javasoft.com/products /jdk /idl/docs/javaIDL-specification.html, 1997.

[Takashio 92] - Takashio, K. e Tokoro, M. "DROL : An Object-Oriented Programming Language for Distributed Real-Time Systems", OOPSLA'92, pp 276-294, 1992.

[Takashio 93] - Takashio, K. e Tokoro, M. "Type Polymorphic Invocation: A Real-Time Communication Model for Distributed Systems", Technical Reports, Department of Computer Science, Keio University - Japan, 1993.

[Takashio 96] - Takashio, K., Tokoro, M. "Least Suffering in Distributed Real-Time Programming Language DROL", Real-Time System, 11, pp. 41-70, 1996.

[Tokoro 93] - Tokoro, M e Takashio, K. "Toward Languages and Formal Systems for Distributed Computing", Technical Reports, Department of Computer Science, Keio University - Japan, 1993.

[Tokuda 89] - Tokuda, H. e Mercer, C. W. "ARTS: A Distributed Real-Time Kernel", Operating Systems Review, vol. 23, n. 3, pp 29-53, julho 1989.

[Tomlinson 89] - Tomlinson, C. e Singh, V. "Inheritance and Synchronization with Enabled-Sets", OOPSLA'89 Proceedings, pp. 103-112, outubro 1989.

[Vrchoticky 94] - Vrchoticky, A. "The Basis for Static Execution Time Prediction", Dissertaation, Techische Universität Wien, abril 1994.

[Yokote 92] - Yokote, Y. "The Apertos Reflective Operating Systems: The Concept and its Implementation", OOPSLA'92 Proceedings, pp. 414-434, 1992.

[Yonezawa 86] - Yonezawa, A. et al. "Object-Oriented Concurrent Programming in ABCL/1", SIGPLAN Notices, vol. 21, n. 11, novembro 1986.

[Wahl 94] - Wahl, T. and Rothernel, K. "Representing Time in Multimedia Systems", In Proc. of International Conference Multimedia Computing and Systems, pp. 538-543, 1994.

[Watanabe 88] - Watanabe, T. and Yonezawa, A. "Reflection in an Object-Oriented Concurrent Language", OOPSLA'88 Proceedings, pp. 306-315, setembro 1988.

[Zhang 93] - Zhang, N. et. al. "Pipelined Processors and Wors-Case Executin Time", Journal of RealTime Systems, october, 1993.

Apêndice A - Especificação sintática de Java/RTR

Como visto no capítulo IV, Java/RTR é uma extensão de Java destinada a representar explicitamente as construções temporais definidas pelo modelo RTR. Neste apêndice a sintaxe das extensões introduzidas é especificada formalmente, via extensão da gramática LALR(1) usada na especificação formal da sintaxe de Java [Sun 95a]. Para representar as extensões introduzidas, utilizaremos a mesma notação empregada em [Sun 95a]; segundo essa notação, símbolos não-terminais são escritos em estilo "itálico", enquanto que símbolos terminais são escritos em estilo "normal". Adicionalmente, usaremos o estilo "negrito" para destacar as produções (regras sintáticas) introduzidas e as alterações das regras existentes.

A.1 - Declaração de classes e meta-classes

ClassDeclaration :

Modifiers$_{opt}$ ***RTROption***$_{opt}$ class *Identifier Super*$_{opt}$ *Interface*$_{opt}$ *ClassBody*

RTROption : one of

RTBC MMC SMC CMC

A.2 - Declaração de campos (RT-Type e Path-Expression)

FieldDeclaration :

Modifiers$_{opt}$ *Type VariableDeclarator* ;

RTTypeDeclaration ;

PathExpressionDeclaration ;

RTTypeDeclaration :

RT-Type ***Identifier*** = ***RTType***$_{opt}$ (***FormalParameterList***)

RTType :

"TT" ,

"ET" ,

PathExpressionDeclaration :

path ***PathExpression*** **end**

PathExpression :

PathExpression | ***TermPathExpression***

TermPathExpression

TermPathExpression :

TermPathExpression ; ***FactorPathExpression***

FactorPathExpression

FactorPathExpression :

Identifier

(PathExpression)

Literal : (PathExpression)

A.3 - Declaração de métodos

MethodDeclaration :

MethodHeader MethodBody

MethodHeader :

Modifier$_{opt}$ *Type MethodDeclarator Throws*$_{opt}$ ***TimingConstraint***$_{opt}$ ***TimingExceptionHandler***$_{opt}$ ***CategoryClause***$_{opt}$

Modifier$_{opt}$ void *MethodDeclarator Throws*$_{opt}$ ***TimingConstraing***$_{opt}$ ***TimingExceptionHandler***$_{opt}$ ***CategoryClause***$_{opt}$

TimingConstraint :

RTIdentifier (TCAtributeList),

TCAtributeList :

TCAtribute

TCAtributeList , TCAtribute

TCAtribute :

Identifier

Identifier = Literal

TimingExceptionHandler :

Identifier (ArgumentList)

CategoryClause :

category = *CategoryName*

CategoryName :

Identifier

StringLiteral

A.4 - Ativação de métodos

StatementExpression :

Assignement

PreIncrementExpression

PreDecrementExpression

PosIncrementExpression

PosDecrementExpression

MethodInvocation

@ MethodInvocation

ClassInstanceCreationExpression

PrimaryNoNewArray :

Literal

This

(*Expression*)

ClassInstanceCreationExpression

MethodInvocation

@ MethodInvocation

ArrayAccess

MethodInvocation :

Name (*ArgumentList*$_{opt}$) ***TimingParameters***$_{opt}$ ***TimeoutClause***$_{opt}$

Primary . *Identifier Name* (*ArgumentList*$_{opt}$) ***TimingParameters***$_{opt}$ ***TimeoutClause***$_{opt}$

super . *Identifier* (*ArgumentList*$_{opt}$) ***TimingParameters***$_{opt}$ ***TimeoutClause***$_{opt}$

TimingParameters :

, **(** ***ArgumentList*** **)**

TimeoutClause :

timeout **(** ***Expression*** **)** **,** **{** ***TimeoutException*** **}**

TimeoutException :

case timeout : ***TimeoutHandlerException RejectException***$_{opt}$ ***AbortException***$_{opt}$

RejectException :

case reject : ***TimeoutHandlerException***

AbortException :

case abort : ***TimeoutHandlerException***

TimeoutHandlerException :

Block

Identifier **(** ***ArgumentList*** **)**

Apêndice B - Representação dos operadores de intervalo no modelo RTR

Como exposto no capítulo III, os 10 operadores do modelo de intervalos estendido usados na representação das relações de sincronização multimídia, podem ser implementados pelo modelo RTR através da associação de restrições temporais básicas (*ActivationInterval, Periodic, Aperiodic* e *Start-at*) aos métodos responsáveis pela apresentação das mídias. A sincronização desejada será obtida pela ativação assíncrona (representada pelo símbolo @) destes métodos e pelo fornecimento de valores aos atributos das restrições temporais utilizadas.

Neste apêndice, por questão de uniformidade, representaremos todos os operadores considerando que os métodos responsáveis pela apresentação das mídias (métodos *A()* e *B()*) possuem um intervalo de ativação associado (restrição temporal *ActivationInterval*). Contudo, enfatizamos que esta representação é apenas uma das possibilidades existentes; dependendo da aplicação e do contexto no qual a sincronização deve ser obtida, diferentes restrições temporais e/ou diferentes esquemas de ativação podem vir a ser utilizados.

B.1 - Declaração dos métodos *A()* e *B()*

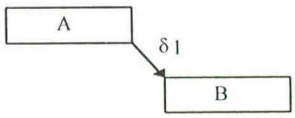
```
...
void A( ... ), ActivationInterval ( Inicio, Fim )
{ ... }
void B( ... ), ActivationInterval ( Inicio, Fim )
{ ... }
...
```

B.2 - Operador *before*

- Relação de sincronização :

  A *before* (δI) B

- Representação gráfica :



- Implementação RTR :

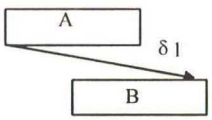
```
...
@A( ... ), (I1, F1);
@B( ... ), (F1+δ1, F2);
...
```

B.3 - Operador *beforeendof*

- Relação de sincronização :

A *beforeendof* (δ1) B

- Representação gráfica :



- Implementação RTR :

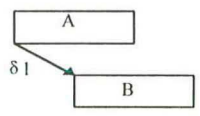
```
...
@A( ... ), (I1, F1);
@B( ... ), (I2, I1+δ1);
...
```

B.4 - Operador *cobegin*

- Relação de sincronização :

A *cobegin* (δ1) B

- Representação gráfica :



- Implementação RTR :

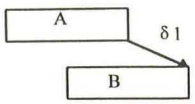
```
...
@A( ... ), (I1, F1);
@B( ... ), (I1+δ1, F2);
...
```

B.5 - Operador *coend*

- Relação de sincronização :

A ***coend*** ***(δ1)*** **B**

- Representação gráfica :



- Implementação RTR :

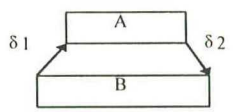
```
...
@A( ... ), (I1, F1);
@B( ... ), (I2, F1+δ1);
...
```

B.6 - Operador *while*

- Relação de sincronização :

B ***while*** ***(δ1, δ2)*** **A**

- Representação gráfica :



- Implementação RTR :

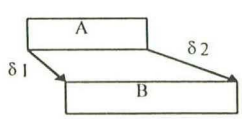
```
...
@B( ... ), (I1, F1);
@A( ... ), (I1+δ1, F1-δ2);
...
```

B.7 - Operador *delayed*

- Relação de sincronização :

A ***delayed*** ***(δ1, δ2)*** **B**

- Representação gráfica :



- Implementação RTR :

...

@A(...), (I1, F1);

@B(...), (I2+δ1, F1+δ2);

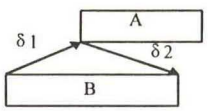
...

B.8 - Operador *startin*

- Relação de sincronização :

B ***startin*** *(δ1, δ2)* **A**

- Representação gráfica :



- Implementação RTR :

...

@B(...), (I1, I1+δ1+δ2);

@A(...), (I1+δ1, F2);

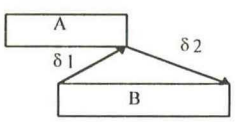
...

B.9 - Operador *endin*

- Relação de sincronização :

A ***endin*** *(δ1, δ2)* **B**

- Representação gráfica :



- Implementação RTR :

...

@A(...), (I1, F1);

@B(...), (F1-δ1, F1+δ2);

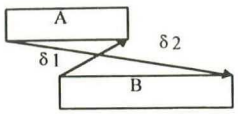
...

B.10 - Operador *cross*

- Relação de sincronização :

A ***cross(δ1, δ2)*** **B**

- Representação gráfica :



- Implementação RTR :

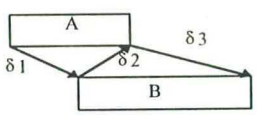
```
...
@A( ... ), (I1, I2+δ1);
@B( ... ), (I2, I1+δ2);
...
```

B.11 - Operador *overlaps*

- Relação de sincronização :

A ***overlaps (δ1, δ2)*** **B**

- Representação gráfica :



- Implementação RTR :

```
...
@A( ... ), (I1, I1+δ1+δ2);
@B( ... ), (I1+δ1, I1+δ1+δ2+δ3);
...
```